Vagner Barbosa do Nascimento

Modelagem e geração de interfaces dirigidas por regras

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Daniel Schwabe

Rio de Janeiro

Abril de 2013

Vagner Barbosa do Nascimento

Modelagem e geração de interfaces dirigidas por regras

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Daniel Schwabe
Orientador
Departamento de Informática - PUC-Rio

Prof. Edward Hermann Haeusler
Departamento de Informática - PUC-Rio

Prof[a]. Simone Diniz Junqueira Barbosa
Departamento de Informática - PUC-Rio

José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico - PUC-Rio

Rio de Janeiro, 15 de abril de 2013

Vagner Barbosa do Nascimento

Graduou-se em Informática com Ênfase em Análise de Sistemas pela Universidade Estácio de Sá em 2007. Atuou como analista de sistemas, coordenador de projetos de tecnologia, desenvolvedor de aplicações web e sistemas embarcados. Possui interesse acadêmico e profissional nas áreas de engenharia de software, hipertexto e multimídia, linguagens de programação e tecnologias para Web Semântica.

Ficha Catalográfica

Nascimento, Vagner Barbosa do

Modelagem e geração de interfaces dirigidas por regras / Vagner Barbosa do Nascimento ; orientador: Daniel Schwabe. – 2013.
150 f. : il. (color.) ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2013.
Inclui bibliografia

1. Informática – Teses. 2. Interfaces. 3. Adaptação. 4. Regras. 5. Desenvolvimento dirigido por modelos. 6. Web semântica I. Schwabe, Daniel. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Dedico esse trabalho à minha esposa Lucimila.

Agradecimentos

Ao meu orientador Professor Daniel Schwabe pela confiaça e incentivo para vencer os desafios.

À Profa. Simone Barbosa pelos ensinamentos e ao Prof. Edward Hermann por ter atendido prontamente a minha solicitação de participação da banca de avaliação desta dissertação.

A todos do laboratório GIGA / PUC-Rio pelo incentivo contínuo e ao Prof. Raul Nunes pelo apoio incondicional para que me decicasse a esse trabalho.

Ao meu grande amigo Mauricio Bomfim que desde sempre me estimulou a não desistir e realizar essa conquista.

Meu pai, mãe, irmão e a minha família que nunca duvidou da minha capacidade, tanto os parentes mais próximos quanto à querida família chilena.

E em especial a minha amada esposa Lucimila, testemunha do meu esforço.

Resumo

Nascimento, Vagner Barbosa do; Schwabe, Daniel. **Modelagem e geração de interfaces dirigidas por regras**. Rio de Janeiro, 2013.150p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Hoje em dia é incontável o volume aplicações desenvolvidas para a W*orld Wide Web*. Essas aplicações possuem interfaces com o usuário que devem ser capazes de se adaptar a diversas situações de uso, mudanças de contexto e conteúdo e ainda ser compatíveis com vários navegadores e dispositivos. Além disso, o projeto e a manutenção de interfaces que necessitam de adaptações em função das regras de negócio da aplicação demandam boa parte do esforço exigido durante do ciclo de vida do desenvolvimento de uma aplicação. Para auxiliar no projeto dessas interfaces, algumas UIDL's (*User Interfaces Description Languages*) foram propostas com o intuito de oferecer um nível de abstração para que o projetista não precise focar a atenção em aspectos mais concretos durante o desenvolvimento de uma interface. Esse trabalho apresenta uma proposta para modelagem e geração de interfaces de aplicações web baseadas em regras de produção. Essas regras definem critérios para as situações de: acionamento de uma interface, seleção dos elementos que participam da composição abstrata e do mapeamento dos *widgets* concretos que serão utilizados na etapa de renderização. Essa proposta contempla um método para modelagem das interfaces, uma arquitetura de implementação e um ambiente de autoria e execução dos modelos de interface. Também será apresentada uma arquitetura para construção de *widgets* concretos, uma máquina de interpretação e renderização de interfaces. O objetivo geral da proposta é conseguir projetar interfaces mais sensíveis aos dados e aos contextos de uso, cobrir certas situações de adaptação e gerar interfaces mais flexíveis e reutilizáveis.

Palavras-chave

Interfaces; Adaptação; Regras; Desenvolvimento Dirigido por Modelos; Web Semântica.

Abstract

Nascimento, Vagner Barbosa do; Schwabe, Daniel (Advisor). **Rule-based approach to modeling and generation user interfaces**. Rio de Janeiro, 2013. 150p. MSc. Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Today there is a countless number of applications developed for the World Wide Web. These applications have user interfaces that should be able to adapt to several usage scenarios, content and context changes and also to be compatible with multiple browsers and devices. Furthermore, the design and maintenance of interfaces that need adjustments depending on the business rules of the application require much effort during the development life cycle of an application. In order to assist in the design of these interfaces, some UIDL's (User Interface Description Languages) have been proposed aiming at providing a level of abstraction so that the designer does not need to immediately focus attention on concrete aspects during the development of an interface. This work presents a proposal for modeling and generating interfaces of web applications based on production rules. These rules define criteria for situations determining the activation of an interface; for the selection the elements that participate in the abstract composition and also for the mapping of specific widgets that will be used in the rendering stage. The proposal contemplates a method for modeling interfaces, an implementation architecture and a framework for authoring and execution of the proposed interface models. An architecture is also presented for building widgets as well as a concrete interface interpretation and rendering machine from a hierarchy specification. The overall goal of the proposal is to design interfaces more responsive to data and contexts of use, including situations of adaptation, generating more flexible and reusable interfaces.

Keywords

Interfaces; Adaptation; Rules; Model Driven Development, Semantic Web.

Sumário

Lista de figuras

Lista de quadros

Lista de tabelas

1
Introdução

No início da década de 1990, Myers e Rosson [1992] relataram que o esforço para desenvolver interfaces com o usuário nas aplicações consumia 48% do tamanho do código de fonte, 45% do tempo de desenvolvimento, 50% do tempo de implementação, e ainda, 37% do tempo total da manutenção das aplicações.

Desde então, esses valores possivelmente aumentaram devido à evolução das plataformas de computação, o advento da Internet e da WWW e ao surgimento das mais recentes e populares modalidades de interface com interação gestual e vocal.

Com isso, os desenvolvedores de interfaces precisam lidar com diferentes aspectos e ambientes heterogêneos para:

- Múltiplas plataformas de computação: desktops, laptops, tablets, smartphones, dispositivos embarcados; uma grande variedade de modalidades de interação: digitação, voz, sensores de movimento, (multi) toque, etc.
- Conjuntos múltiplos de tarefas em evolução que devem ser apoiadas, derivando um número crescente de diferentes formas de trabalho (*workflow*) e essas devem ser suportadas pelo aplicativo;
- Perfis de usuários diversos, que variam de novatos a especialistas, de culturas diferentes e falando uma infinidade de idiomas.

Além disso, o contexto de uso, geralmente formado pela tríade <usuário, plataforma, ambiente>, muda dinamicamente enquanto o aplicativo está sendo utilizado, demandando a chamada plasticidade das interfaces [Coutaz e Calvary, 2012], para que a interface se adapte ao novo contexto, preservando a "experiência do usuário".

A Modelagem de Interfaces Dirigidas por Modelos[1] tem sido utilizada para lidar com os desafios apresentados, tentando manter o esforço de desenvolvimento e manutenção de interfaces, sobre tais condições, em níveis toleráveis.

Iniciativas como o Cameleon Framework Reference (CFR) [Meixner et al., 2011] servem como referência para projetar e classificar métodos e arquiteturas de interfaces baseadas em modelos, levando em conta o suporte a diversos contextos de uso e também a questão da plasticidade das interfaces.

1.1. Objetivos

Esta dissertação apresenta um método de modelagem de interfaces dirigidas por regras que visa abordar certos aspectos de adaptação, tornando as interfaces mais sensíveis aos dados e os modelos mais aderentes aos princípios do Cameleon Framework Reference. Além disso, também será apresentada uma arquitetura, um ambiente de autoria e uma máquina de interpretação e geração de interfaces de aplicações web projetadas com esses modelos.

O método e a arquitetura propostos nesse trabalho são independentes de metodologia ou ambientes de projeto de aplicações web, podendo assim ser utilizados e analisados livremente. Contudo, na apresentação desse trabalho o método está sendo tratado com uma evolução dos modelos de interface existentes no SHDM - *Semantic Hypermedia Design Method* [Lima, 2003].

1.2. Organização da dissertação

Os temas apresentados nessa dissertação estão organizados em capítulos da seguinte maneira:

- **Capítulo 2 – Fundamentos**: uma breve revisão das principais áreas de conhecimento envolvidas no desenvolvimento desse trabalho;
- **Capítulo 3 – Método de modelagem de interfaces**: descreve de forma sucinta o método de modelagem proposto;

[1] Do inglês Model Based User Interface (MBUI)

- **Capítulo 4 - Arquitetura de interpretação e geração de interfaces**: apresenta o funcionamento da arquitetura proposta, da máquina de execução e do ambiente de autoria;
- **Capítulo 5 - Exemplos:** traz alguns exemplos práticos de interfaces e seus modelos para uma aplicação;
- **Capítulo 6 - Conclusão:** traz uma breve avaliação das capacidades e limitações detectadas, sugere alguns trabalhos futuros e contribuições trazidas a partir desse trabalho.

2
Fundamentos

2.1.
Abordagens para modelagem de interfaces

2.1.1.
User Interfaces Description Languages (UIDL's)

No contexto do desenvolvimento de interfaces com o usuário dirigido por modelos, é de grande importância a evolução das chamadas UIDL's (*User Interface Description Languages*). Uma UIDL consiste numa linguagem de alto nível que permite descrever as características relevantes de uma interface gráfica com o usuário [MENDES, 2009]. A princípio, esse tipo de linguagem não exige conhecimentos de programação, permitindo que analistas, designers, programadores e o usuário final possam utilizá-la durante o ciclo de vida do desenvolvimento.

Sendo assim, várias linguagens têm sido criadas e cada uma aborda diferentes aspectos de uma interface como: portabilidade, independência de dispositivos, suporte a múltiplas plataformas, etc.

A maioria das UIDL's desenvolvidas até então foram descritas em notação XML (*eXtensible Markup Language*) e [GARCIA et al., 2009], num trabalho comparativo de tais linguagens faz o seguinte relato em relação ao volume de UIDL's existentes[2]:

> A ampla disponibilidade de linguagens de marcação e a capacidade de introduzir qualquer linguagem baseada em XML, juntamente com a multiplicidade de plataformas disponíveis hoje, despertou esta corrida e exacerbaram-na a um ponto em que hoje mais do que uma dúzia de UIDLs existem com foco em algumas das características desejadas.

[2] Outros trabalhos comparando as UIDL's existentes foram publicados, como por exemplo, o de POHJA [2010], [GARCIA et al., 2009] e [MENDES, 2009].

Podemos dividir as UIDL's existentes em dois grupos: as **fundamentadas em modelos** e as **não fundamentadas em modelos**. Porém, nesse trabalho, o foco foi verificar as características das UIDL's fundamentadas em modelos, no que também é conhecido como MDUI (*Model Driven User Interfaces*).

Como exemplo de UIDL's não fundamentadas em modelos, podemos considerar:

- **XUL**[3] (Mozilla): construir aplicações ricas em recursos multi-plataforma que podem ser executadas conectadas ou desconectadas da Internet;
- **OpenLaszlo**[4] (Laszlo) – construir aplicações ricas na internet utilizando a linguagem declarativa LZX. A interface gerada é um binário executável Flash(. swf)[5];
- E ainda Microsoft **XAML**[6] e Adobe **MXML**[7].

2.1.1.1. HTML5

Apesar de ser considerada apenas uma linguagem de marcação, a versão 5 do HTML (*Hypertext Markup Language*), chamada de HTML5[8] merece a devida atenção. O HMTL5 ainda não é suportado completamente por todos os navegadores de internet e dispositivos móveis até o momento. Porém, representa uma grande evolução em relação às versões predecessoras do HTML. Podemos destacar suas principais diferenças:

- Elementos *MathML*[9] e *SVG*[10] incluídos diretamente no documento, possiblitando a renderização programática de imagens em 2D;
- Novos tags para marcação semântica da estrutura da página como, por exemplo: section, article, aside, header, footer, etc;

[3] https://developer.mozilla.org/en-US/docs/XUL
[4] http://www.openlaszlo.org/
[5] http://pt.wikipedia.org/wiki/Adobe_Flash
[6] http://msdn.microsoft.com/en-us/library/ms752059.aspx
[7] http://pt.wikipedia.org/wiki/MXML
[8] http://www.w3.org/TR/html5/
[9] http://www.w3.org/TR/MathML3/
[10] http://www.w3.org/Graphics/SVG/

- Execução nativa de conteúdo multimídia, como áudio e vídeo;
- Novos tipos de campos de formulários para suportar entradas específicas, tais como email, data, data e hora, cores, busca, números, url, etc.;
- Novos atributos para tratar certos comportamentos, tais como validação de campos de formulário, edição de valores em loco (*Edit-in-place*), arrastar e soltar (*drag in drop*), etc.

2.1.2. Interfaces Dirigidas por Modelos

Nos últimos anos, diversas abordagens têm sido desenvolvidas para se enfrentar os desafios existentes na geração de interfaces com o usuário, muitas delas através do uso de modelos[11]. Porém, os desafios mais recentes estão relacionados em oferecer mecanismos para se projetar interfaces que possam atender as diversas plataformas, dispositos e variados contextos de uso[12]. Além disso, Thevenin e Coutaz [1999], e mais tarde, J.Coutaz e Calvary [2012] apresentam a questão da *plasticidade das interfaces*, como sendo a capacidade delas se adaptarem aos contextos de uso preservando a *usabilidade* e os *valores humanos*.

2.1.2.1. Cameleon Reference Framework

Publicado em 2002 através do Projeto CAMELEON, o CAMELEON Reference Framework (CRF) é a referencia mais aceita para projetar e classificar métodos e arquiteturas de interfaces baseadas em modelos, levando em conta o suporte a diversos contextos de uso e também a questão da plasticidade das interfaces.

Um contexto de uso em geral é formado por três entidades:

[11] Para uma melhor noção histórica da evolução das MBUI's, recomenda-se a leitura do trabalho de Meixner, Paternó e Vanderdonckt [2011].

[12] Grupos de trabalhos como o Model-Based UI XG e o Model-Based UI Working Group foram formados para avaliar o uso e a cobertura oferecida por MBUI's.

- **Usuário** – incluir atributos que definem o perfil do usuário ou o usuário que utiliza um sistema;
- **Plataforma** - descreve a parte computacional, como por exemplo, sistema operacional, características físicas e de software (dispositivo, aplicativo, recursos, etc) ;
- **Ambiente** – Inclui atributos relacionados ao espaço (lugar), tempo, funções e outras propriedades que caracterizem quando e onde a interação está ocorrendo. Por exemplo, localização do usuário, som ambiente, iluminação, as redes de dados corrente, etc.

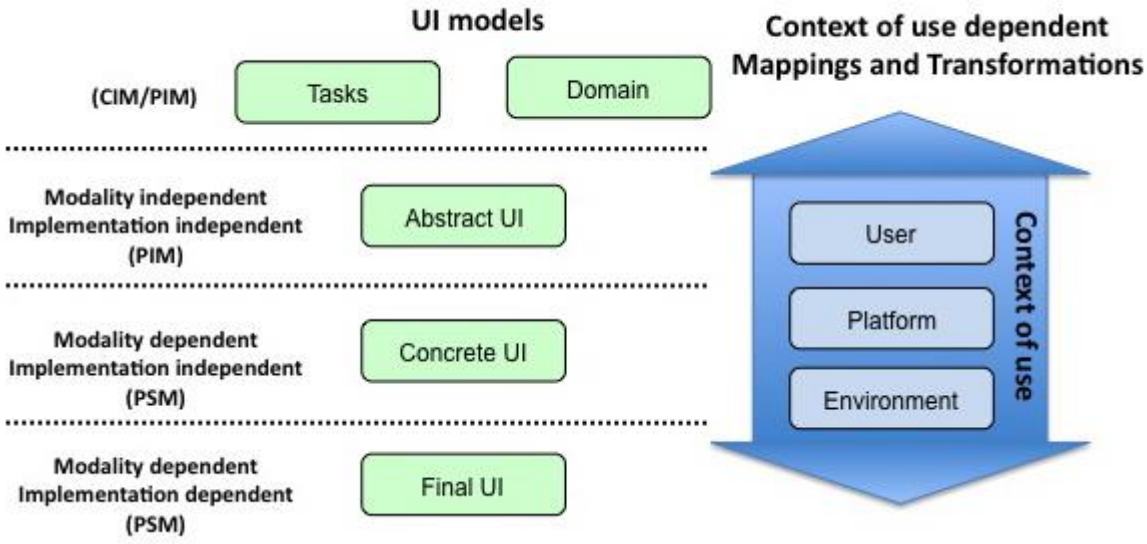

Figura 1 – Diagrama simplificado do CRF [Calvary et al., 2002]

Em relação aos modelos existentes no CRF, a Figura 1 ilustra cada um deles e a influência do contexto de uso < Usuário, Plataforma, Ambiente> sobre cada etapa de modelagem, sendo essas:

- **Tarefas e domínio** - definem que objetos de domínio podem ser manipulados com uma interface de usuário e que tarefas que podem ser realizada pelo usuário (por exemplo, 'comprar uma passagem');
- **Interface Abstrata (AUI)** - define os elementos de interface que estão previstos em um nível abstrato, sem focar em que componente concreto irá ser utilizado. Por exemplo, 'um elemento para selecionar um valor'. Esses elementos são independentes de implementação ou plataforma;

- **Interface concreta (CUI)** - define quais *widgets* concretos que possam satisfazer a tarefa. Por exemplo, "*Widget de controle deslizante*" ou "*um campo de entrada de texto validado*". Os *widgets* concretos são dependentes do ambiente de execução;
- **Interface final** - é a interface final gerada e utilizável por um usuário. Essa pode ser executável como um código binário, ou ainda, interpretada em tempo de execução, como por exemplo, em HTML.

Através das etapas que compoem o CRF, diversas UIDL's existentes puderam ser avaliadas e classificadas quanto à sua cobertura e nível de abstração oferecido em seus artefatos. Um exemplo disso é o trabalho de avaliação feito por [PAULHEIM, 2011], onde a Figura 2 apresenta diversas UIDL's por ele analisadas e a cobertura de cada uma sobre a perspectiva das etapas do CRF.

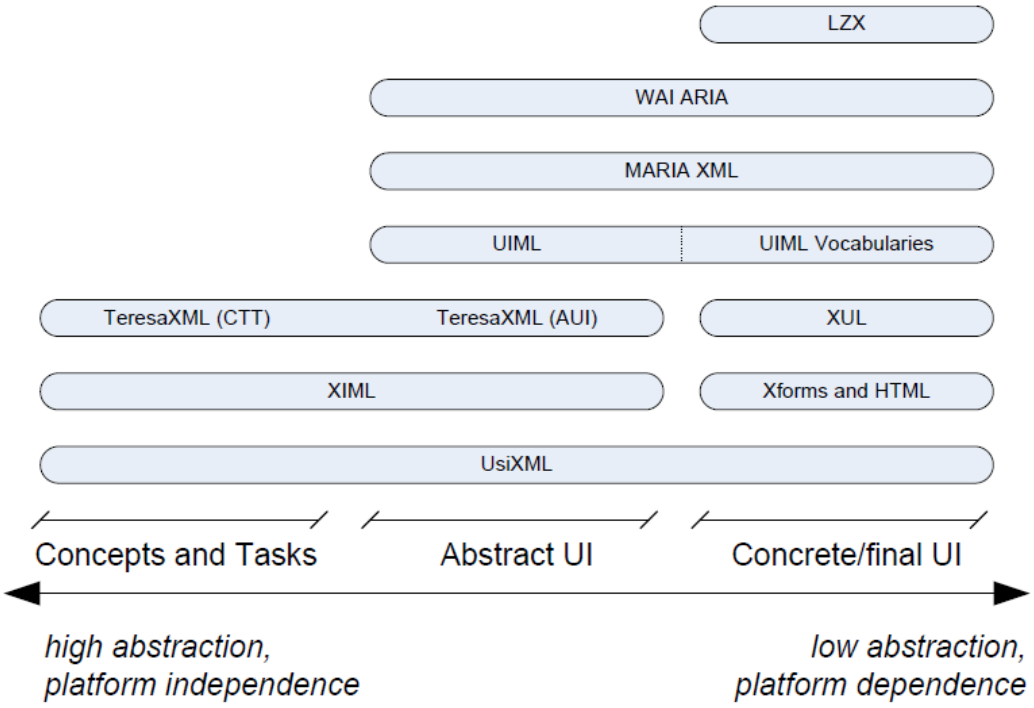

Figura 2 – Visão geral sobre diversas UIDL's e cobertura de cada através do CRF

Conforme comentado anteriormente, existe uma grande variedade de UIDL's para os mais diversos propósitos, como por exemplo, o UIML [HELMS et al., 2009], o UWE [KOCK et al., 2008] e o WebML [CERI et al., 2000]. Porém não é papel desse trabalho relatar todos, compará-los ou avaliá-los, já que

existem trabalhos realizados com foco nesses aspectos[13]. Porém, por serem de maior relevância para esse trabalho, os projetos TERESA, MARIA e USIXML serão comentados a seguir, além do SHDM que será apresentado na seção 2.3.

2.1.2.2. TERESA / MARIA XML

A TERESA XML foi desenvolvida pelo projeto TERESA, o qual está integrado ao projeto Europeu (Cameleon IST) [BERTI et al., 2004]. A ferramenta TERESA XML dá suporte à especificação dos modelos até a geração das interfaces para múltiplas plataformas. A Figura 3 ilustra a correspondência entre os esquemas representados em XML, desde o modelo de tarefas CTT (ConcurTaskTrees), a modelagem abstrata e a concreta.

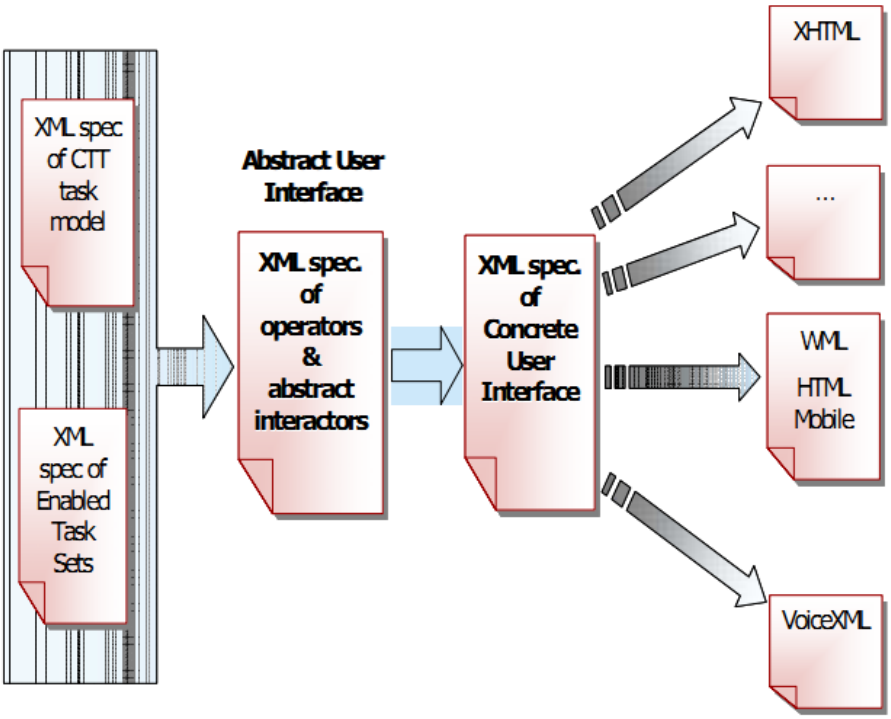

Figura 3 – Correspondência entre modelos nas estapas de modelagem no TERESA

Modelo de Tarefas

A modelagem no TERESA é feita num esquema *top-down*, começando num nível mais alto, desenvolvendo um modelo único para capturar os possíveis

[13] Conforme já mencionado nas notas de rodapé 2 e 12.

contexto de uso e os papéis envolvidos. Também é desenvolvido um modelo de domínio que visa identificar os objetos que terão de ser manipulados pelas tarefas realizadas e os relacionamentos entre os objetos. Esse modelo é especificado usando a notação do *ConcurTaskTrees*[14] (CTT), onde é possível derivar do XML do CTT os *Presentation Task Sets* (PTSs) que são conjuntos de tarefas que ocorrem no mesmo período de tempo.

Modelo de Interface Abstrata

A partir dos *Presentation Task Sets* é possível especificar os modelos abstratos de interface, onde uma interface é composta por objetos do tipo *presentation*, que são definidos por:

- **Connection** - que está relacionado ao comportamento dinâmico da *presentation*, além da navegação para outro *presentation*;
- **Interactor** - interação entre o usuário e a aplicação, podem ser do tipo *Selection, control* e *edit*.

Modelo de Interface Concreta

Com o modelo de interface abstrato definido pode-se modelar a interface de concreta, num processo dependente de plataforma. Todavia, propriedades do dispositivo para qual a interface será gerada devem ser consideradas.

Com isso, é possível gerar interfaces do tipo: XHTML[15] para computadores pessoais e dispositivos móveis, além de diálogos *VoiceXML*[16].

MARIA XML

O projeto MARIA [PATERNO et al., 2009] é a evolução do projeto TERESA, que traz a capacidade de projetar e desenvolver interfaces multi-dispositivo, além de dar suporte a interfaces migratórias, para aplicações baseadas em *WebServices*.

[14] http://hiis.isti.cnr.it/tools/CTT/home

[15] http://www.w3.org/TR/xhtml1

[16] http://www.w3.org/TR/voicexml20/

Resumidamente, podemos listar que as principais características suportadas foram:

- Suporte a interação contínua e fluxo de entrada paralelo (non-WIMP apps);
- Incluído um modelo de dados para uso das interfaces abstratas e concretas (XSL[17]);
- Incluído um modelo de eventos, para definir como a interface responde a eventos do usuário (*trigger*);
- Atualização contínua de campos (AJAX[18] para web interfaces, *callback* para aplicações standalone, mecanismos de acesso assíncrono, etc);
- Atualização dinâmica de partes da UI;

2.1.2.3. USIXML

A UsiXML *(User Interface eXtensible Markup Language)* consiste numa linguagem que permite a especificação de interfaces com usuário independentemente da linguagem de programação e plataforma em que irá ser executada [LIMBOURG, 2004]. Assim a USIXML dá suporte a múltiplos contextos de utilização, como *Character User Interfaces* (CUIs), *Graphical User Interfaces* (GUIs), *Auditory User Interfaces*, além de suporte a interfaces multimodais [MENDES, 2009].

A sintaxe é definida por um conjunto de esquemas XML, em que cada esquema corresponde a um dos tipos de modelo existentes na linguagem. Assim, a distribuição dos esquemas da USIXML entre as chamadas *etapas de transformação* proporciona cobertura para todos os modelos previstos no Cameleon Reference Framework.

[17] http://www.w3.org/Style/XSL/

[18] Asynchronous JavaScript and XML

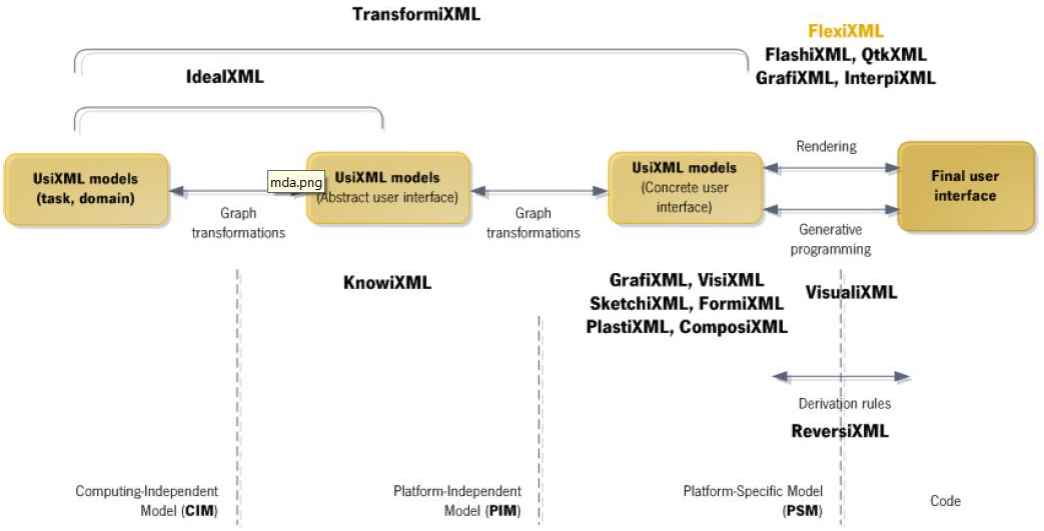

Figura 4 – Diferentes ferramentas que dão cobertura ao USIXML sobre o CRF [VANDERDONCKT, 2005]

Porém, diferentemente do projeto TERESA, a USIXML possui diferentes ferramentas para dar suporte aos modelos de domínio e tarefas, a definição dos modelos abstratos e concretos e a geração e execução das interfaces propriamente ditas, conforme apresentado na Figura 4.

Ainda comparando com a ferramenta TERESA, os modelos de interface concreta não são diretamente dependentes da plataforma de execução, podendo esses serem adotados para diferentes contextos de utilização.

Até o momento a USIXML é a UIDL com maiores avanços do ponto de vista de cobertura para os diversos aspectos adotados pelo CRF, contendo o conjunto mais completo de esquemas para especificação dos modelos de interfaces e interação com o usuário.

2.2. Sistemas de regras de produção

Sistemas de regras de produção ou simplesmente *sistemas de produção* são sistemas computacionais muito utilizados em sistemas baseados em conhecimento.

> Esses sistemas são utilizados quando a formulação do problema a ser resolvido é complexa e quando existe uma grande quantidade de conhecimento específico do domínio sobre como resolvê-lo. Normalmente uma boa indicação a respeito do uso desta tecnologia é a existência de um especialista humano capaz de solucionar o problema.
>
> [REZENDE, 2003]

Diferente de uma codificação comum, uma representação do conhecimento deve levar em conta algumas características: ser compreensível ao ser humano, abstrair detalhes do funcionamento interno e ser robusta, funcionando mesmo que não comporte todas as situações existentes [DAVIS et al., 1993].

Por esse motivo, sistemas de produção são amplamente utilizados em Inteligência Artificial para modelar comportamentos inteligentes e construir sistemas especialistas [GUPTA & FORGY, 1983].

Tais sistemas são construídos basicamente por uma **base de conhecimento** e **uma máquina de inferência**, conforme ilustrado na Figura 5. A base de conhecimento é composta por *regras*, também chamadas de produções, e por *fatos*.

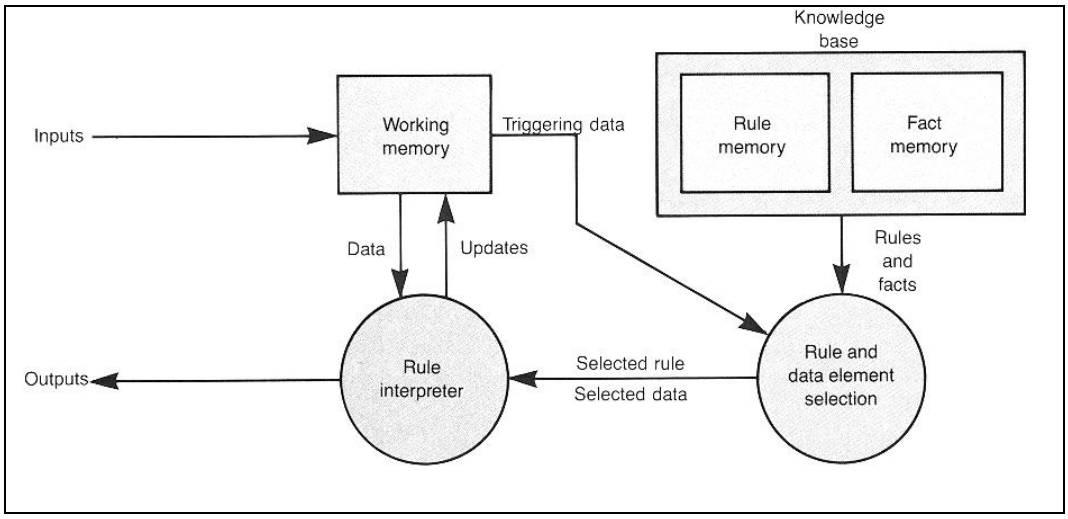

Figura 5 – Componentes básicos de um sistema de produção [ROTH, 1985]

As regras são expressas em duas partes: uma *condição* (se) e uma *consequente* ou *ação* (então). Essas partes são conhecidas também como *lado esquerdo* e *direito da regra*. Sendo assim, uma regra pode ser interpretada de maneira que, caso a *condição* seja satisfeita, a *consequente* também será. Caso a consequente seja uma ação, o resultado é que a ação será executada. Alguns sistemas também possuem uma *memória de trabalho* que armazena o conhecimento, ou seja, novos fatos que forem sendo adquiridos durante a interpretação das regras [ROTH, 1985].

Para exemplificar o funcionamento da interpretação de regras, imaginemos que uma base de conhecimento (BC) possua os seguintes fatos:

P1 - < *Téo* é **filho** de *João* >

P2 - < *Pedro* é **filho** de *João* >

E a seguinte regra fosse avaliada:

Se < P1 é **filho** de X > e < P2 é **filho** de X> *então*

<P1 e P2 são **irmãos**>

Assim um novo conhecimento seria gerado, acrestando na memória de trabalho um novo fato que indica que <*Téo* e *Pedro* são **irmãos**>. Esse novo fato será considerado na avaliação das regras subseqüentes.

O último e não menos importante componente de um sistema de produção é *intepretador de regras* ou *máquina de inferência*. Esses mecanismos podem utilizar dois métodos de raciocínio para as regras:

- **Encadeamento progressivo** (*forward chaining*) - *Parte dos fatos* na BC e na memória de trabalho, buscando quais regras eles satisfazem, para produzir assim novas conclusões (fatos), ou ainda, realizar ações, até que a hipótese seja incluída na base de conhecimento ou uma condição de parada seja alcançada;
- **Encadeamento regressivo** (*backward-chaining*) - *Parte da hipótese* que se quer provar, procurando regras na base de conhecimento cujo *consequente* satisfaz essa hipótese. Ou seja, buscando regras na BC que possam responder as perguntas. Com isso, apenas regras relevantes às perguntas serão avaliadas, buscando como resultado provar que a hipótese inicial é verdadeira.

O algoritmo de encadeamento progressivo mais conhecido e utilizando nas implementações de sistemas de produção é *o algoritmo Rete*[19], que foi concebido por Forgy [1979]. Foi utilizado inicialmente como base do sistema de produção OPS5 e mais tarde nos sistemas CLIPS[20], Jess[21], Drools[22], JRules[23], OPSJ[24], Blaze Advisor[25] e BizTalk Rules Engine[26].

[19] http://en.wikipedia.org/wiki/Rete_algorithm

[20] http://en.wikipedia.org/wiki/CLIPS

[21] http://en.wikipedia.org/wiki/Jess_programming_language

[22] http://www.jboss.org/drools

Existe uma grande diversidade de sistemas de produção e máquinas de inferência para os mais distintos propósitos. A seguir serão apresentados alguns exemplos para ilustrar alguns casos de uso.

2.2.1. JBoss Drools

JBoss Drools é uma plataforma integrada para manipular regras de lógica negócio. O Drools é organizado em diversos projetos, estando a máquina de inferência no *Drools Expert*, que é uma implementação em Java do algoritmo de Rete.

As regras no Drools são compostas de um *nome*, das *condições* e *ações* a serem executadas, conforme exemplo do Quadro 1.

```
rule "My Rule"
  when // Lado esquerdo
    Person(name == "John") //Condições
  then //Lado direto
    System.out.println("Hi John!"); //Ações
end
```

Quadro 1 – Formato em que as regras são declaradas no Drools

2.2.2. Euler Proof

Euler Proof[27] é uma máquina de inferência construída para dar apoio a provas lógicas. Utiliza método de encadeamento regressivo adicionado de um mecanismo de detecção de caminho euleriano[28]. Existem implementações em Java, C#, Python, Javascript e Prolog.

A notação utilizada para representar os fatos é Notation3, ou simplesmente N3[29], que é uma notação que estende o meta-modelo do RDF[30], adicionando variáveis, implicações lógicas e predicados funcionais. Já as regras são expressas

[23] http://www-01.ibm.com/software/websphere/products/business-rule-management/

[24] http://www.pst.com/opsj.htm

[25] http://www.fico.com/en/Products/DMTools/Pages/FICO-Blaze-Advisor-System.aspx

[26] http://www.microsoft.com/biztalk/en/us/business-rule-framework.aspx

[27] http://eulersharp.sourceforge.net/GUIDE

[28] http://pt.wikipedia.org/wiki/Caminho_euleriano

[29] http://www.w3.org/TeamSubmission/n3/

[30] http://en.wikipedia.org/wiki/Resource_Description_Framework

em N3Logic[31]. A idéia é que isso torne possível expressar regras no ambiente da web.

O Quadro 2 ilustra um pequeno exemplo de regra expresso em N3Logic.

```
@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@keywords.
@forAll x, y, z.
{
  x parent y. y sister z
} log:implies {
  x aunt z
}
```

Quadro 2 – Exemplo de regra em N3Logic

O sistema de regras de produção utilizado nesse trabalho, o Wongi-Engine, será comentado na seção 4.2.1.2, por se tratar de componente chave da implementação da arquitetura realizada[32].

2.3. SHDM

O método SHDM (*Semantic Hypermedia Design Method*), proposto originalmente por [Lima, 2003], é uma abordagem baseada em modelos para projetar aplicações hipermídia. O SHDM é uma evolução do método OOHDM e, por esse motivo, manteve os seus fundamentos, enriquecendo cada etapa com mecanismos inspirados nas linguagens propostas para a Web Semântica [MOURA, 2004].

2.3.1. Etapas

O SHDM é composto por seis etapas: *Levantamento de Requisitos, Modelagem do Domínio, Projeto Navegacional, Projeto de Interface, Projeto Comportamental e Implementação*.

[31] http://www.w3.org/DesignIssues/N3Logic

[32] Não foi considerado nenhum algoritmo de encadeamento regressivo no desenvolvimento desse trabalho simplesmente por não ser aplicável ao problema que se quer tratar. Não são inseridas na base de conhecimento regras cujo termo **consequente** indique que interface deve ser selecionada ou que elementos da interface devem ser considerados, nunca satisfazendo tal hipótese.

Cada etapa produz um ou mais modelos focados em distintos aspectos da aplicação modelada, conforme ilustrado na Tabela 1.

	Artefato	Descrição	Etapa
1	Descrição dos cenários e casos de uso	Identificação dos atores e tarefas apoiadas pela aplicação.	Levantamento de Requisitos
2	UIDs	Diagramas de interação do usuário.	Levantamento de Requisitos
3	Ontologias de domínio	Vocabulários para definição das instâncias de domínio (recursos RDF). Pode ser qualquer ontologia da Web Semântica definida em OWL ou RDFS.	Modelagem de Domínio
4	*Namespaces*	Conjunto de *namespaces* (pares de prefixos e URIs) utilizados pela aplicação.	Modelagem de Domínio
5	Repositórios	Repositórios de dados da Web Semântica, geralmente acessados via SPARQL *Endpoints*. A aplicação usa os dados disponíveis nestes repositórios como instâncias do domínio.	Modelagem de Domínio
6	Instâncias do domínio	Dados do domínio da aplicação, definidos segundo a ontologia de domínio. Mais especificamente, recursos RDF.	Modelagem de Domínio
7	Mapeamento navegacional	Descrição dos atributos navegacionais nas classes de domínio Esquemas de contextos, estruturas de acesso e *landmarks*. Definidos com um vocabulário específico do método.	Projeto Navegacional
8	Modelo Navegacional	As instâncias do modelo navegacional são recursos RDF enriquecidos com os atributos navegacionais. Sua geração pode ser em tempo de execução.	Projeto Navegacional
9	Modelo de operações	Definição das operações da aplicação que dão semântica as regras de negócio e apoio as tarefas de interação com o	Projeto Comportamental

		usuário. Definidas com um vocabulário específico do método.	
10	Modelo de Interfaces	Definição de elementos da interface abstrata, descrições retóricas e seus mapeamentos para o modelo de operações, incluindo o navegacional, e para componentes da interface concreta. Definida com um vocabulário específico do método	Projeto de Interface
11	Ontologia de componentes da interface concreta	Definição de possíveis componentes da interface concreta para uso na implementação. Definida com um vocabulário específico do método.	Projeto de Interface

Tabela 1 – Artefatos do SHDM

Vale ressaltar que o fruto do presente trabalho substitui os artefatos anteriores de modelagem de interfaces apresentados no quadro acima (itens 10 e 11). Novos mecanimos de modelagem abstrata e concreta foram propostos e esses serão apresentados no capítulo 3.

Criado por Bomfim [2011], o Synth é um ambiente de desenvolvimento que dá suporte à construção de aplicações segundo o SHDM, fornecendo um conjunto de módulos capazes de receber como entrada os modelos gerados na execução das etapas do método e produzir como saída uma aplicação hipermídia descrita por estes modelos. O Synth foi utilizado como ambiente de autoria de aplicações para o desenvolvimento deste trabalho, sendo comentado na seção 4.2.1.1 e apresentado um exemplo executável no capítulo 5.

3
Método de modelagem de interfaces dirigidas por regras

Apesar de substituir os artefatos existentes no SHDM para modelagem de interfaces, o método aqui apresentado herda princípios da Ontologia de Widgets Abstratos e da Ontologia de Widgets Concretos, que são artefatos originalmente propostos por [Moura, 2004]. Mais tarde, o método foi estendido por [Luna, 2009] com o intuito de expressar interfaces da Web 2.0, que ficou conhecido como interfaces RIA (Rich Internet Aplication) [Bozzon et al., 2006].

A proposta desse trabalho, além de contemplar a modelagem abstrata e o mapeamento concreto, propõe o uso de um sistema de regras de produção em etapas distintas da modelagem, com o objetivo de:

- Atender a mudanças no contexto de uso <Usuário, Plataforma, Ambiente>, fornecendo mecanismos de adaptação através de regras, tornando assim as interfaces mais sensíveis aos dados;
- Dar maior independência as interfaces, transferindo para cada uma a responsabilidade de decidir se é a mais adequada a uma requisição do usuário;
- Possibilitar que a interface consiga definir uma melhor composição em função dos dados recebidos, selecionando os elementos mais adequados para se compor;
- Tornar o mapeamento abstrato $\Rightarrow$ concreto mais flexível, sendo possível para um mesmo elemento abstrato mapear *widgets* distintos, numa relação zero-para-muitos, ou ainda, declarar múltiplas configurações dos parâmetros de renderização para um mesmo elemento;
- Tornar as interfaces mais flexíveis, estimulando o reuso das mesmas para diferentes casos de navegação;
- Oferecer um método mais aderente às especificações das MBUIs atuais.

Além disso, a proposta amplia significativamente a responsabilidade dos *widgets* concretos, que podem ser tratados como artefatos de software auto-contidos (caixa-preta), sendo esses responsáveis por gerar o código dos elementos concretos de interação com o usuário e também encapsular os comportamentos de cada elemento. As seções 3.4.1 e 4.2.8 fornecem maiores detalhes sobre a construção dos *widgets* concretos.

Mesmo herdando princípios da modelagem de interfaces do SHDM e o fato deste trabalho ter sido desenvolvido convenientemente aproveitando a arquitetura existente e os demais modelos do SHDM, o método e a arquitetura aqui propostos podem ser analisados e utilizados de forma independente em outros ambientes de projeto e execução de aplicações web.

Sucintamente as etapas propostas para o projeto de interface segundo o método são:

- A declaração de regras de seleção de interface;
- A modelagem da composição da interface abstrata;
- A declaração das regras de seleção de elementos abstratos;
- A declaração das regras de mapeamento concreto;
- Definição de extensões de renderização concreta.

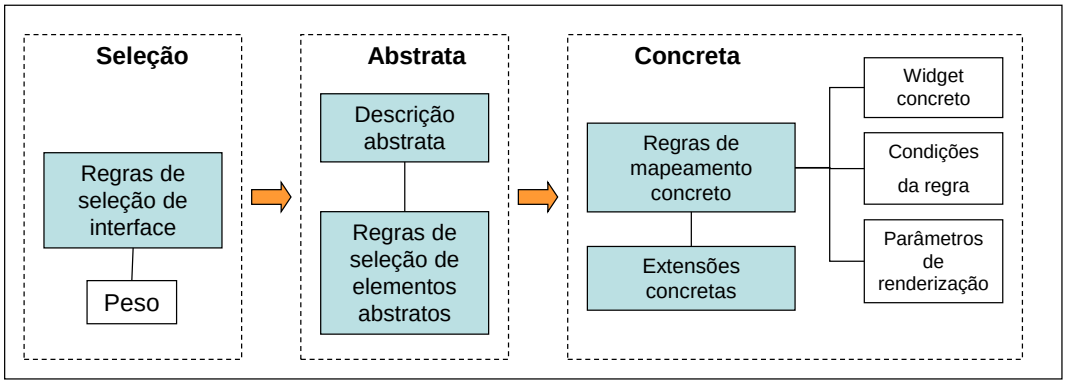

Figura 6 - Etapas de modelagem de interface

3.1. Entrada de dados da interface

Para uma melhor compreensão das etapas seguintes, é importante esclarecer quais são os conjuntos de dados mais comuns que uma interface precisa receber como entrada, tanto para avaliar as regras quanto para renderizar a interface propriamente dita, levando em conta o método proposto. Sendo assim, devem ser considerados como entrada de dados de uma interface os seguintes conjuntos:

- **Dados de regras** – utilizados como um conjunto de premissas (fatos) para as regras de produção e utilizadas na avaliação de todos os elementos e etapas que compõem uma interface. Quaisquer informações disponíveis e processáveis podem ser utilizadas como fatos a serem avaliados pelas regras. Como exemplo de dados de regras podem ser citados:
    - Propriedades, tipos de dados e literais de um determinado elemento navegacional em questão;
    - Parâmetros de navegação recebidos numa requisição;
    - Cabeçalhos do navegador de internet (User-Agent[33]);
    - Dados do dispositivo e plataforma utilizada;
    - Variáveis disponíveis no ambiente;
    - Outras informações capturadas.
- **Dados de mapeamento** – são os dados que serão utilizados na etapa de mapeamento concreto, servindo como entrada dos parâmetros para renderização concreta e também para compor as condições das regras de seleção abstrata e de mapeamento concreto. Como exemplo pode ser citado:
    - Dados de um nó navegacional em questão;
    - Coleções de dados iteráveis para geração de índices ou listas;
    - Parâmetros de navegação;
    - Qualquer outra informação que precise ser renderizada na interface.

Os tópicos a seguir dão mais detalhes sobre a utilização desses dados, além do capítulo 4 que fornece mais informações sobre a arquitetura desenvolvida.

3.2. Regras de seleção de interfaces

Como ponto de partida da modelagem de uma interface[34], o primeiro passo será declarar as regras de seleção da interface em questão, com o objetivo principal de definir em que situação a interface será ser acionada.

[33] Campo cabeçalho do protocolo HTTP - http://en.wikipedia.org/wiki/User_agent

Através desse mecanismo o projetista poderá especializar o uso de uma determinada interface, ou ainda, flexibilizar o seu acionamento para mais de uma situação, contanto que a interface projetada comporte atender a diversos contextos de utilização.

Seguem abaixo alguns exemplos de caso de uso prático:

a. A interface só será acionada para exibir um determinado índice de contexto;
b. A interface pode exibir qualquer índice de contexto;
c. A interface só é acionada caso os parâmetros *'start-date'* e *'end-date'* forem fornecidos;
d. Uma interface só será acionada quando o usuário estiver autenticado pela aplicação;
e. A interface só é válida para dispositivos móveis.

Dessa maneira pode-se imaginar uma grande variedade de situações em que seleção por alguns critérios é desejada.

No caso de uma aplicação que possua mais de uma interface, o que provavelmente é o caso mais comum, certos conflitos são esperados, uma vez que as regras de mais uma interface podem ser válidas para os mesmos dados de entrada fornecidos. Nos exemplos citados acima, os casos *a* e *b* podem ser conflitantes, já que o caso *b* atende a qualquer índice de contexto. A solução adotada para desempate dos conflitos foi a atribuição de **pesos** para indicar qual é a ordem de prioridade que cada interface deve ser avaliada. Quanto menor o peso (mais leve)m maior a prioridade no momento da avaliação.

As seções 4.1 e 4.2.4 trazem mais informações sobre a arquitetura desenvolvida para seleção de interfaces e os termos de uma DSL (*Domain Specific Language*) para descrição das regras.

3.3. Modelagem de composição da interface abstrata

A modelagem de interface abstrata descreve os possíveis elementos que representam as interações entre o usuário e a aplicação. A escolha desses elementos (*widgets*) é realizada pelo designer, através do modelo navegacional e da análise de

[34] Entenda-ee interface como 'interface com o usuário' que é acionada como resposta de uma operação a uma requisição do usuário.

> requisitos, visto que a partir deles é possível se extrair todas as necessidades de trocas de informação que precisam acontecer entre o usuário e aplicação.
>
> [Moura, 2004]

Conforme definido por Moura [2004], a modelagem abstrata procura separar os aspectos mais essenciais da composição da interface, independentes da tecnologia e arquitetura de implementação.

O método apresentado nesse trabalho adota os conceitos abstratos do vocabulário definido na *Ontologia de Descrição de Interfaces Ricas* [Luna, 2009]. Também foi adicionada uma camada de regras para selecionar quais elementos devem participar da composição abstrata.

3.3.1. Widgets Abstratos

A interface abstrata representa o nível mais elementar dos elementos perceptíveis da interação entre o usuário e a aplicação. Tais elementos estão organizados de maneira hierárquica e formam uma composição abstrata. Cada elemento representante dessa composição é classificado conforme sua funcionalidade, sendo também chamado de *widget* abstrato.

Os termos utilizados para representar a classificação de cada elemento estão presentes na *Ontologia de Descrição de Interfaces Ricas*, proposta por [Luna, 2009], sendo uma evolução da *Ontologia de Widgets Abstratos* presente no SHDM [Moura, 2004]. Todavia, as demais classes presentes na ontologia[35] proposta por Luna não foram consideradas nesse trabalho e algumas definições dos *widgets* também foram modificadas.

Assim, todos os *widgets* abstratos existentes são subclasses da classe **AbstractInterfaceElement**, estando disponíveis os seguintes termos para classificar os elementos de uma composição abstrata:

- **AbstractInterface**: representa o elemento raiz de uma interface abstrata, agrupando os demais elementos da hierarquia.
- **CompositeInterfaceElement**: representa uma composição de outros elementos abstratos;

[35] Não foram consideradas as classes ConcreteInterfaceElement, RhetoricalStructure, Transition e Event, e consequentemente suas subclasses.

- **ElementExhibitor**: representa elementos que exibem qualquer tipo de conteúdo;
- **IndefiniteVariable**: representa elementos que permitem a entrada de dados não arbitrados;
- **PredefinedVariable**: representa elementos que permitem a seleção de dados arbitrados, ou seja, de valores pré-definidos;
- **SimpleActivator**: representa qualquer elemento capaz de acionar alguma funcionalidade, navegação ou evento.

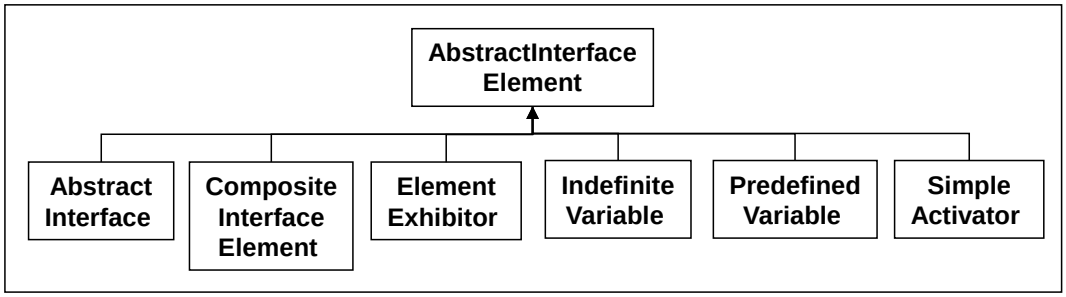

Figura 7 – Classe AbstractInterfaceElement e Widgets Abstratos disponíveis

3.3.1.1. Modelando uma interface abstrata

Para modelar uma interface abstrata, o projetista, utilizando os *widgets* abstratos apresentados na seção 3.3.1, deve organizar hierarquicamente, em uma estrutura de *árvore*, os elementos que vão compor a interface projetada, onde cada nó da árvore é classificado por um *widget* abstrato.

Cada nó participante da hierarquia projetada deve possuir duas propriedades:

- **name**: atribui uma identificação única ao elemento em questão na composição abstrata. Essa propriedade é obrigatória;
- **repeatable**: informa se os elementos filhos do nó em questão são repetíveis, ou seja, representam uma coleção de dados, uma lista, índice, etc. Essa propriedade é opcional.

Durante essa etapa, como sugestão para o projetista, recomenda-se ter em mente apenas aspectos relacionados à composição e funcionalidade, deixando de lado questões da exibição concreta como: aparência, posicionamento e possíveis

comportamentos que cada elemento deverá possuir. Essas questões só devem ser consideradas na etapa de mapeamento concreto.

Segue abaixo um exemplo hipotético de composição abstrata, sem utilizar nenhuma notação específica. Todavia, a seção 4.2.6 descreve uma notação em *Hash Ruby*[36] utilizado na arquitetura de execução desenvolvida nesse trabalho.

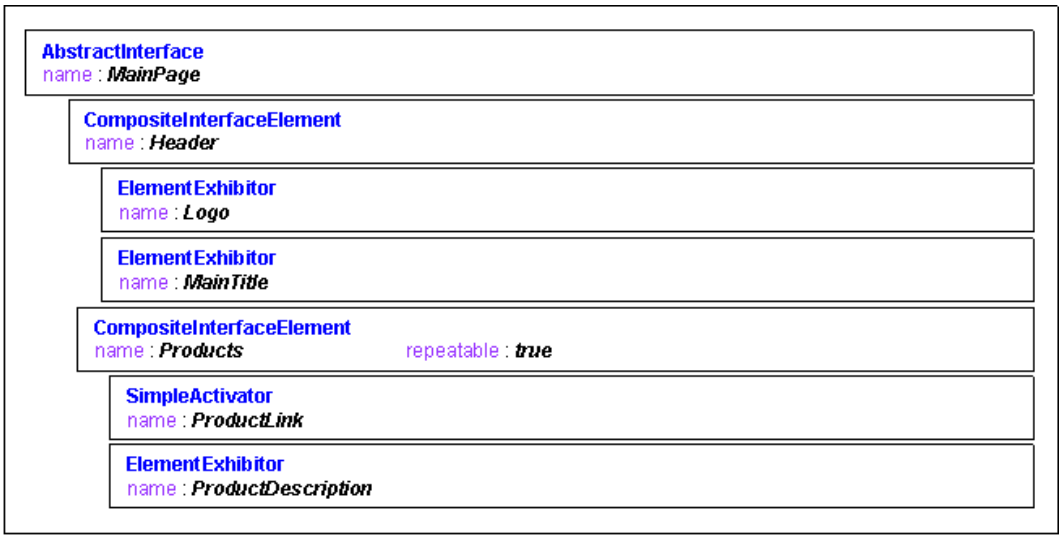

Figura 8 – Exemplo de composição abstrata

A Figura 8 demonstra um exemplo simples de composição abstrata de uma interface que exibe uma lista de produtos com ativadores para navegar para outro contexto. O exemplo também demonstra o uso da propriedade *repeatable* para indicar que os elementos internos são repetíveis, ou seja, uma coleção de produtos.

3.3.2. Regras de seleção de elementos abstratos

A seção 3.3.1.1 expõe um exemplo de interface modelada através do uso de *widgets* abstratos. Entretanto, imaginando-se que a interface representada no exemplo da Figura 8 fosse o resultado de uma busca por produtos. Supõem-se dois casos:

- *Caso normal* - uma coleção de produtos foi localizada e serão exibidos como resultado da busca;
- *Caso alternativo* - nenhum produto foi localizado pela busca.

[36] http://www.ruby-doc.org/core-1.9.3/Hash.html

Nesse último caso, a composição projetada na Figura 8 não contempla elementos abstratos suficientes para, por exemplo, informar que *nenhum produto foi localizado*.

Situações como a citada acima podem ser atendidas fornecendo para a interface a capacidade alterar a sua composição em função de critérios fornecidos pelo projetista.

Outras questões também justificam a alteração da composição abstrata das interfaces como:

- Ajustar a composição em função dos dados recebidos para exibição;
- Possibilitar a reutilização de uma interface para casos similares de composição;
- Fazer certas adaptações da interface ou de porções dela em função de mudanças no contexto de uso, cobrindo certos casos estudados nas chamadas *interfaces adaptativas*[37];
- Expressar certas condições ligadas a atender regras de negócio da aplicação.

Como continuidade do método de modelagem de interface, uma etapa de descrição de **regras de seleção de elementos abstratos** é proposta como abordagem para fornecer para as interfaces a capacidade de alterar a sua composição em função das regras e dos dados de entrada fornecidos.

[37] http://en.wikipedia.org/wiki/Adaptive_user_interface. Não foi realizada nenhuma avaliação de quais níveis de adaptação são suportados pelo mecanismo de regras de seleção tanto abstrato quanto concreto, ficando como sugestão para trabalhos futuros.

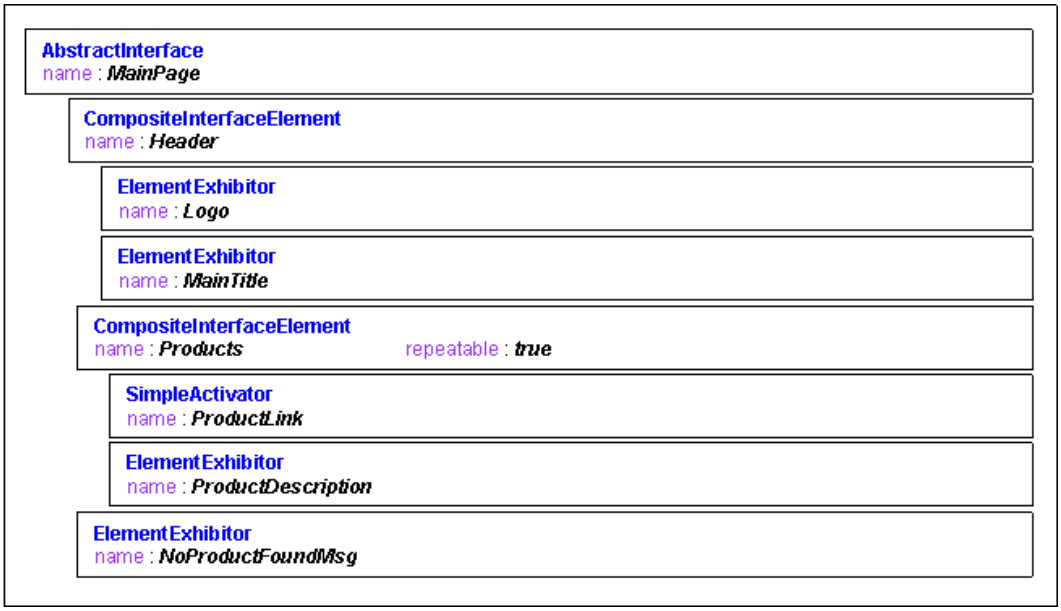

Figura 9 – Interface abstrata com elemento para produtos não encontrados

Para exemplificar o uso de regras de seleção de elementos abstratos, recorre-se novamente ao exemplo da Figura 8, porém agora com algumas inclusões.

A Figura 9 recebeu em sua composição o elemento *NoProductFoundMsg*, que representa exatamente o elemento exibidor da mensagem de que "nenhum produto foi encontrado".

Para que a seleção dos elementos ocorra conforme o desejado, certas condições devem ser definidas. A Figura 10 exemplifica as condições em que cada elemento deve ser selecionado, e assim, considerado ou descartado da composição abstrata.

Figura 10 - Regras que condicionam a seleção dos elementos Products e NoProductFoundMsg

A regra 1 indica que o elemento *Products* só será incluído na composição se lista de produtos não for vazia. A regra 2 indica que o elemento *noProductFoundMsg* só será incluído se a condição for exatamente a oposta da regra 1, caso a lista seja vazia.

Para leitura e definição das regras, algumas premissas podem ser consideradas:

- Caso algum elemento da composição não possua regras, ele será avaliado como válido na composição, sendo assim incluído;
- Um elemento da composição possuindo regras:
  - Sendo avaliada como *verdadeira,* o elemento será incluído na composição abstrata;
  - Sendo avaliada como *falsa*, o elemento será descartado;
- Os elementos hierarquicamente inferiores (filhos) a um elemento que possui regras também terão a inclusão na interface condicionada em função do elemento superior (pai);
- As regras podem ser agrupamentos com conjunções ou disjunções em suas sentenças.

O pequeno exemplo apresentado anteriormente utiliza, para compor as regras, uma lista de produtos dada como entrada da interface. Porém conforme apresentado na seção 3.1, qualquer outra informação passada como entrada da interface pode participar da descrição de uma regra, criando assim condições mais ricas.

A seção 4.2.6.2 traz mais detalhes do uso de regras de seleção na arquitetura desenvolvida, utilizando um mecanismo com regras de produção e apresenta uma DSL para descrição das regras.

3.4. Mapeamento concreto

A terceira etapa do método proposto é o mapeamento concreto. Conforme visto na seção 3.3, um elemento de uma interface abstrata tem a capacidade de representar somente o nível mais elementar dos elementos perceptíveis da interação entre o usuário e a aplicação. Todavia, somente na etapa de mapeamento concreto será possível indicar que elemento concreto será capaz de materializar e reproduzir as funcionalidades de um determinado elemento perceptível da interface.

Os trabalhos de [Moura, 2004] e [Luna, 2009] contemplam um mecanismo de mapeamento abstrato x concreto. O mapeamento é feito através da propriedade

"*maps_to*", onde cada elemento do modelo abstrato indica um elemento concreto presente na *Ontologia de Widgets Concretos* para renderizá-lo.

A proposta apresentada continua fazendo uso dos *widgets* concretos para renderizar os elementos de interface, porém, considerando as seguintes premissas:

- Um elemento abstrato poderá ser mapeado para um ou mais *widgets* concretos;
- O mapeamento de um *widget* concreto é feito através da regra, que sendo esta satisfeita, o mapeamento será concretizado;
- Os *widgets* concretos devem ser tratados simplesmente como artefatos de software auto-contidos e independentes;
- A renderização dos *widgets* concretos é realizada através de parâmetros fornecidos;

Conforme as premissas citadas anteriormente, identificam-se quatro componentes que participam de um mapeamento: o elemento abstrato, as regras, a identificação do *widget* concreto e os parâmetros de renderização.

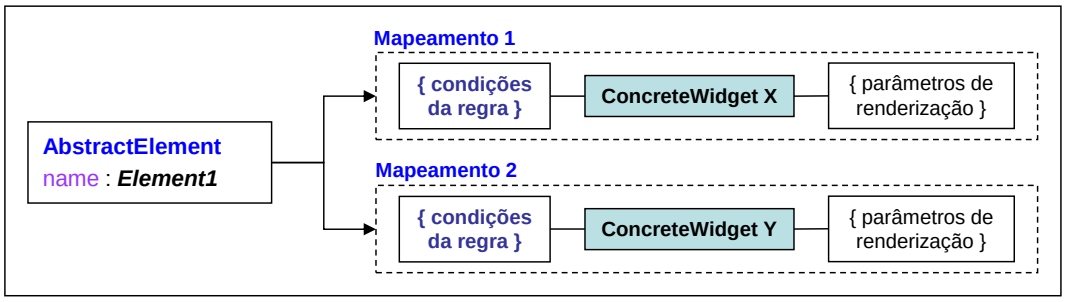

Figura 11 – Esquema de mapeamento abstrato x concreto

A Figura 11 apresenta o esquema de mapeamento proposto, indicando a possibilidade de mapeamento para mais de um *widget* concreto um mesmo *widget* abstrato.

Esses componentes participantes do mapeamento serão apresentados a seguir, além de um mecanismo de extensões de renderização concreta.

3.4.1. *Widgets* concretos e parâmetros de renderização

Conforme comentado anteriormente, nesse trabalho os *widgets* concretos são tratados como artefatos de software, que não precisam necessariamente estar catalogados em uma ontologia específica. Contudo necessitam estar disponíveis no ambiente de modelagem concreta utilizado pelo projetista.

Uma das razões para os *widgets* concretos serem vistos nessa perspectiva é que tanto a tecnologia empregada na renderização das interfaces quanto na construção dos *widgets* evolui rapidamente, o que torna qualquer conjunto proposto obsoleto em pouco tempo. Dessa maneira, novos *widgets* poderão ser construídos regularmente e os existentes podendo ser atualizados conforme a evolução tecnológica.

Seguem abaixo algumas características que podem ser consideradas na construção de *widgets* concretos:

- Devem ser auto-contidos, necessitando apenas receber parâmetros para gerarem os elementos esperados, ou seja, devem ser capazes de se renderizar;
- Devem absorver a complexidade do código concreto gerado, expondo para o projetista apenas suas funcionalidades e capacidades;
- Toda configuração para renderização deve ser feita através de parâmetros expostos para o projetista;
- Caso existam dependências entre outros *widgets*, esses devem possuir a capacidade resolvê-las com baixa intervenção do projetista;
- Devem possuir uma autodescrição com metadados detalhados o suficiente para o projetista compreender como utilizá-lo e/ou o ambiente de execução interpretá-los;
- Os metadados devem descrever com que *widgets* abstratos o *widget* concreto é compatível para mapeamento;
- Os metadados devem descrever quais são os parâmetros suportados pelo *widget*;
- Os *widgets* concretos podem ser distribuídos e disponibilizados de forma individualizada, como ocorre comumente com bibliotecas de software.

Considerando que exista uma coleção razoavelmente completa de *widgets* concretos para as mais variadas funções, a complexidade e a qualidade de uma interface gerada estão diretamente relacionadas com o grau de compreensão do projetista das capacidades dos *widgets* disponíveis e da manipulação dos seus parâmetros. Vale ressaltar que os parâmetros manipulam diretamente propriedades

concretas da renderização, podendo ser valores literais do que será exibido, como por exemplo, classes de uma folha de estilos CSS e o texto (string) de um título.

Outro aspecto é a modelagem de comportamentos, i.e., a resposta a eventos, efeitos, animações, transições e etc., que nesse trabalho são tratadas como capacidades concretas dos *widgets*.[38] Caso o projetista deseje expressar e inserir na interface qualquer uma dessas características citadas, ele deve escolher, para o mapeamento, o *widget* concreto que melhor possa atender a essa necessidade, levando em conta a capacidade do *widget* em fornecê-la e também efetuando ajustes na renderização através dos parâmetros suportados.

Considerando as características e aspectos citados anteriormente, um desenvolvedor de *widgets* concretos pode construir desde *widgets* concretos simples, como por exemplo, para exibir elementos HTML básicos até *widgets* complexos com múltiplas funções, como por exemplo, campos de entrada do tipo calendários dinâmicos, exibidores de dados tipo agenda, listas *drag-and-drop*, etc.

A seção 4.2.8 apresenta uma arquitetura de construção de *widgets* concretos, além de um ambiente de interpretação para esses componentes.

3.4.2. Regras de mapeamento concreto

De maneira análoga as regras de seleção de *widgets* abstratos apresentada na seção 3.3.2, as regras de mapeamento concreto tem o objetivo de selecionar que dupla de ***widget* concreto** e **parâmetros de renderização** será utilizada no mapeamento concreto de cada elemento da composição abstrata.

A razão de se utilizar regras que condicionem a seleção dos *widgets* concretos e os parâmetros é proporcionar configurações distintas de renderização concreta em função da avaliação das regras com os fatos, além da adaptação da interface escolhendo *widgets* concretos mais adequados para atender a mudanças no contexto de uso.

A fim de ilustrar, segue abaixo uma breve lista hipotética de situações que podem ser tratadas com o mecanismo de regras de mapeamento concreto:

---

[38] Apesar dos aspectos citados terem sido tratados como capacidades concretas dos *widgets* concretos, fica como sugestão para trabalho futuro uma análise sobre essa abordagem de encapsulamento.

- Exibir uma lista de produtos e se queira dar um maior destaque somente para o produto de menor preço e nos demais utilizar o estilo padrão;
- Somente utilizar um *widget* de mapa caso o objeto possua coordenadas geográficas;
- A exibição dos itens de série de um automóvel, no formato "legenda do item – características". Caso o automóvel não tenha o item, exibir no lugar de sua característica a mensagem "não possui";
- Utilizar um widget de caixa de seleção (*selectbox*) quando a exibição for para um dispositivo móvel e um botão de opção (*radiobutton*) para outros dispositivos.

Certas situações podem gerar dúvidas ao projetista para decidir aonde declarar as regras. Por exemplo, a regra pode ser declarada na seleção do elemento abstrato ou no mapeamento concreto, já que em certos casos, o resultado desejado poderá ser atingido em uma etapa ou em outra. Porém, cabe o bom senso do projetista de avaliar se:

a. A composição da interface abstrata for alterada, então a regra deve ser aplicada na seleção abstrata;
b. A condição da regra só se aplica a instâncias de uma coleção de dados, ou diretamente a dados concretos que serão exibidos, então a regra deve ser definida no mapeamento concreto.

Para exemplificar possíveis regras de mapeamento concreto, seguem abaixo a Figura 12 e Figura 13, que não fazem uso de uma notação específica e os *widgets* concretos são apenas ilustrativos.

```
maps ProductPrice to widget HTML-Label
        parameters { style : BIGSIZE, content :  product.price }
        condition    { products.price equal product.all-prices.min }

maps ProductPrice to widget HTML-Label
        parameters { style : normal, content :  product.price }
        condition    { }
```

Figura 12 – Regra para condicionar troca de estilos para o menor preço de um produto

A Figura 12 representa o exemplo em que o elemento abstrato *ProductPrice*, será mapeando com o mesmo *widget* concreto *HTML-Label* em qualquer situação, porém caso o preço do produto exibido seja o menor, o parâmetro *style* será definido como *BIGSIZE* e nos demais casos será mapeado como *normal*.

```
maps CategoriesList to widget HTML-SelectBox
        parameters { content : categories.list }
        condition  { user-agent is mobile }

maps CategoriesList to widget HTML-RadioButton
        parameters { content : categories.list }
        condition  { user-agent is not mobile }
```

Figura 13 – Regra para condicionar o mapeamento de um *widget* em função do dispositivo

Na Figura 13 uma lista de categorias será mapeada como *HTML-SelectBox*, caso o usuário esteja visualizando a interface em um dispositivo móvel e, nos demais casos, um *HTML-RadioButton* será utilizado para essa função.

Conforme feito na etapa de regras de seleção de elementos abstratos, algumas **premissas** podem ser consideradas para a melhor compreensão das regras de mapeamento concreto:

- Caso algum elemento da composição não possua regras, ele **não será** mapeado concretamente, não sendo assim renderizado;
- Um elemento da composição possuindo regras:
  - Sendo alguma avaliada como *verdadeira*, o elemento será mapeado e renderizado;
  - Sendo alguma avaliada como *falsa*, ou nenhuma como verdadeira, ele não será mapeado;
- As regras podem ser agrupamentos com conjunções ou disjunções em suas sentenças.
- Um mesmo elemento abstrato pode receber diversas declarações de regras e duplas de *widget* concreto e parâmetros de renderização, sendo avaliados em uma determinada ordem. A primeira declaração válida será utilizada no mapeamento.

Como visto até aqui, múltiplas combinações de *widgets* concretos e parâmetros são possíveis com o mecanismo de regras de mapeamento concreto e

cabe ao projetista decidir qual a melhor combinação para alcançar o resultado de renderização desejado.

A seção 4.2.7 apresenta uma DSL e um ambiente de execução para toda a etapa de modelagem concreta.

3.4.3. Extensões de renderização concreta

Durante a etapa de modelagem abstrata e no mapeamento concreto, o projetista deve tentar considerar todos os elementos que vão compor uma interface e determinar suas funcionalidades através dos *widgets* concretos. Todavia, nem sempre os *widgets* escolhidos têm condições de contemplar todos os comportamentos e realizar a renderização final desejada pelo projetista. Por exemplo, certos comportamentos esperados dependem da integração entre dois *widgets*. Ou ainda, certos elementos adicionais à composição também podem ser necessários durante a renderização.

Com o intuito de estender as capacidades de *widgets* existentes, dar-lhes novos comportamentos, criar ligações entre os *widgets* da composição e inserir novos elementos durante a etapa de renderização, um mecanismo de extensões é proposto para prover essas capacidades adicionais. Outro ponto é que um mecanismo extensões possibilita a existência de uma quantidade menor de *widgets* concretos, além de menos complexos e mais genéricos, já que eles sempre poderão ser estendidos.

Segue uma breve lista de situações para ilustrar casos em que as extensões podem ser desejadas:

- Incluir algum tipo de validação em um campo de formulário;
- Efetuar a cópia de dados de um elemento para outro da interface após um determinado evento ocorrer;
- Condicionar o preenchimento dos dados de uma caixa de seleção em função de outra;
- Inserir um simples espaçador entre dois elementos da interface;
- Inserir um elemento adicional meramente para ajustes visuais da interface, que não foi necessário e contemplado na composição abstrata.

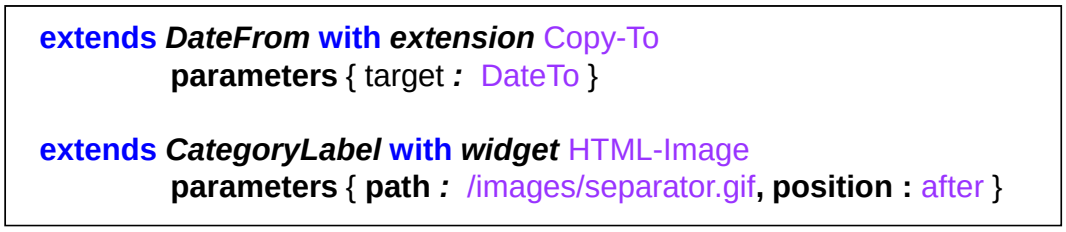

Figura 14 – Exemplo de extensão concreta aplicada a elementos da interface

A Figura 14 traz exemplos hipotéticos da aplicação de extensões a elementos da interface. No primeiro, o elemento *DateFrom* é estendido com a extensão *Copy-To* e, como parâmetro, é informado que o elemento de destino é o *DateTo*. Já o elemento *CategoryLabel* tem inserida uma imagem logo após a sua renderização (parâmetro *position*).

Da mesma maneira que os *widgets* concretos, as extensões concretas devem ser vistas como artefatos de sofware. Cabe também ao projetista compreender seus parâmetros de configuração e suas funcionalidades para melhor expressar os elementos e comportamentos adicionais desejados.

A seção 4.2.7.2 apresenta um mecanismo de extensões concretas e suas possíveis configurações de uso.

4
Arquitetura de interpretação e geração de interfaces

A seção a seguir apresenta uma arquitetura de software, independente de tecnologia de implementação, capaz de receber como entrada a descrição de interfaces, conforme os modelos do método apresentado no capítulo 3 e produzir, como saída, a renderização concreta dessas interfaces.

4.1.
Especificação geral da arquitetura

A Figura 15 apresenta a visão conceitual da arquitetura para aplicações modeladas segundo o método SHDM, integrado ao módulo de interfaces, cuja arquitetura segue as definições do método apresentado no capítulo 3.

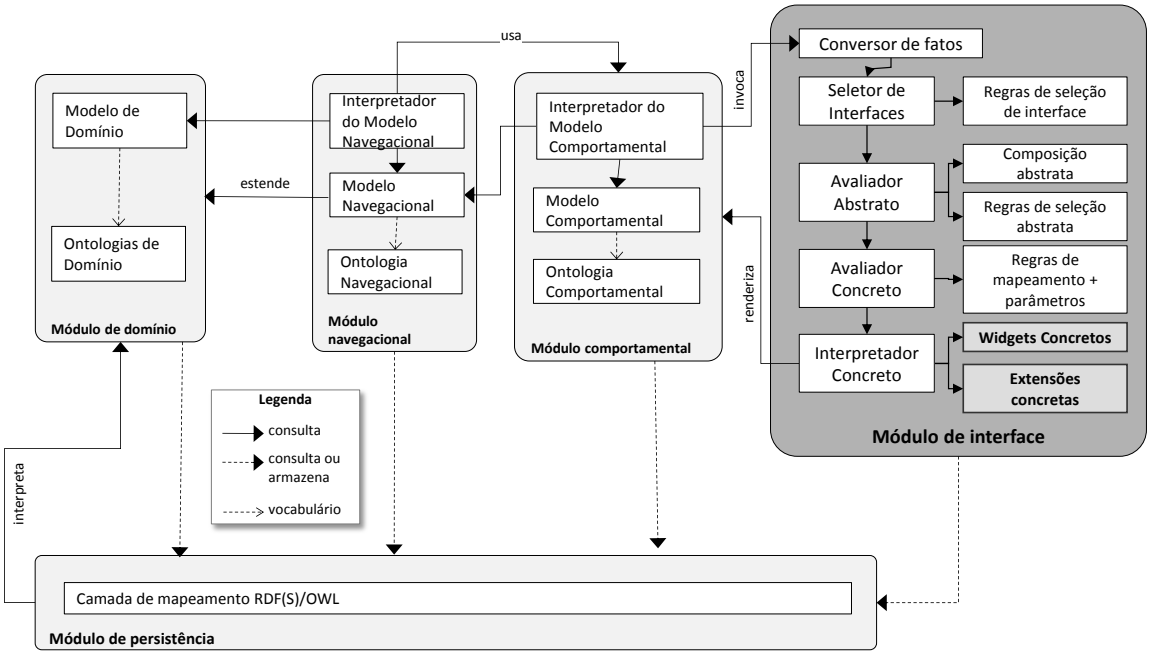

Figura 15 – Visão conceitual da arquitetura do SHDM e módulo de interfaces proposto

A razão pela qual esses modelos foram apresentados juntos foi para ilustrar o uso do método sobre uma arquitetura de modelagem de aplicações existente e também para auxiliar na compreensão do ambiente de implementação utilizado

nesse trabalho. Entretanto, somente os componentes presentes no **módulo de interface** (caixa cinza escuro da Figura 15) serão apresentados mais detalhadamente. Mais detalhes sobre os demais módulos da arquitetura do SHDM podem ser vistos no trabalho de [Bomfim, 2011].

Antes de apresentar os componentes da arquitetura, é importante salientar que, conforme o método apresentado no capítulo 3, as etapas de seleção de interfaces, seleção de elementos abstratos e mapeamento concreto fazem o uso de regras como estratégia de seleção e composição de seus elementos. Por essa razão, como abordagem para a construção dos avaliadores dessas regras, adotou-se nessa arquitetura um **sistema de regras de produção** [GUPTA et al., 1983] para validar as condições descritas nessas etapas. A seção 2.2 traz alguns exemplos de sistemas de produção existentes, não estando essa arquitetura restrita a um sistema específico.

Conforme a Figura 15 os módulos previstos na arquitetura são o de **persistência, domínio, navegacional, comportamental** e o de maior interesse desse trabalho, o de **interface**.

O módulo de persistência trata do acesso e manipulação dos dados da aplicação. Então, além fornecer o acesso aos dados da aplicação, também persiste todos os modelos da aplicação projetada, inclusive os modelos do módulo de interfaces.

O módulo de domínio detém o modelo de domínio da aplicação, que é composta pelas ontologias de domínio e também pelas instâncias de dados da aplicação.

O módulo navegacional mantém as declarações sobre o modelo navegacional, interpreta este modelo e gera o ambiente de execução da navegação do usuário na aplicação [Bomfim, 2011].

O modelo comportamental é composto por operações internas e externas. As internas tratam das regras de negócio da aplicação e não podem ser invocadas por agentes externos, por outro lado, as operações externas podem ser invocadas por agentes externos [Santos, 2010].

As operações externas do modelo comportamental também são as responsáveis por invocar a renderização das interfaces a partir de um agente externo, como por exemplo, a requisição de navegação de um usuário. Para invocar uma interface a partir da execução de uma operação, dois conjuntos de

dados devem ser fornecidos por essa operação: os **dados de regras** e os **dados de mapeamento**. Nesse ponto, a operação externa receberá, do modelo de interface, a interface renderizada, retornando-a para o usuário.

4.1.1. Componentes do módulo de interface

Conforme descrito na seção 3.1, os dados de regras em geral podem ser propriedades de um determinado elemento navegacional, parâmetros de navegação, cabeçalhos do navegador de internet, variáveis disponíveis no ambiente e qualquer outra informação de interesse e que se deseje que sirvam como fatos para as regras das interfaces. Além disso, os dados de mapeamento servirão como entrada dos parâmetros para renderização concreta e também como parte da composição de qualquer regra.

4.1.1.1. Conversor de fatos

Nessa etapa, o módulo de interface repassa os dados de regras para o **conversor de fatos**, que é responsável em converter o conjunto de dados recebido em **fatos** [ROTH, 1985], fazendo a carga inicial de dados na base de conhecimento do sistema de regras de produção (PRS[39]). Os fatos gerados farão parte da base de conhecimento para todas as etapas de avaliação.

4.1.1.2. Seletor de interfaces

Logo após os fatos serem gerados, será acionado o seletor de interfaces. Através do módulo de persistência, cada uma das interfaces será consultada pela ordem de seus pesos atribuídos, sendo a de menor peso consultada primeiro.

Cada interface consultada terá as suas regras incluídas na base de conhecimento do PRS e, junto com os fatos armazenados anteriormente, essas regras serão avaliadas. Caso o resultado da avaliação seja verdadeiro, ou seja, as regras foram correspondidas, a interface será selecionada.

[39] Do inglês Production Rule System.

4.1.1.3. Avaliador abstrato

Com a interface adequada selecionada, o avaliador abstrato consulta o modelo da composição abstrata desta interface, e o conjunto de regras de seleção de cada um dos seus elementos. Com os seguintes passos, o avaliador extrai a composição abstrata resultante:

1. Verifica se a sintaxe fornecida para o esquema abstrato está correta, seguindo a hierarquia de composição, conforme o método apresentado na seção 3.3.1;
2. Inclui as regras de composição abstrata na base de conhecimento da PRS;
3. Executa a avaliação das regras sobre o conjunto de fatos;
4. Para os elementos em que suas regras foram avaliadas positivamente, são gerados novos fatos. Isso significa que esses elementos devem ser incluídos na composição abstrata;
5. O avaliador então descarta da composição **os elementos cujas regras não resultaram em novos fatos gerados**. Ou seja, serão selecionados elementos da composição abstrata em dois casos: os elementos que não possuíam regras declaradas explicitamente e para aqueles em que as regras foram avaliadas como satisfeitas.

4.1.1.4. Avaliador concreto

Após a avaliação da composição abstrata, o módulo de interface repassa a composição resultante para o avaliador concreto. A principal tarefa do avaliador concreto é avaliar as regras de mapeamento e criar **uma estrutura de dados em árvore** válida como entrada para o interpretador concreto. Nessa etapa, as regras fazem uso dos fatos inicialmente inseridos na base de conhecimento da PRS, mas principalmente dos dados de interface que são utilizados nos parâmetros de mapeamento concreto.

Assim, o avaliador concreto precisa percorrer cada elemento da composição abstrata verificando se existem regras de mapeamento concreto para esse elemento, conforme as definições citadas na seção 3.4.2.

Enquanto o avaliador percorre a composição abstrata, também é necessário verificar os elementos que possuem a propriedade *repeatable*, já que nesses casos, uma coleção de dados é recebida como parâmetro. Sendo assim, para cada item da coleção percorrida, são aplicadas as regras existentes para os elementos pertencentes à subárvore hierarquicamente inferior. Cada elemento da subárvore considerado válido na avaliação das regras de mapeamento vai ser incluído na estrutura da árvore final.

A última tarefa é verificar as extensões concretas declaradas para cada elemento da composição e anexá-las aos respectivos elementos (*nós*) da estrutura a ser enviada para o interpretador concreto.

A seção 4.2.7 apresenta uma arquitetura implementada utilizando uma DSL específica, tanto para declarar as regras de mapeamento concreto quanto para as extensões.

4.1.1.5. Interpretador Concreto

O último componente e um dos mais importantes da arquitetura do módulo de interfaces é o interpretador concreto. Ao receber como entrada uma estrutura em árvore, representando os elementos da composição da interface e uma lista de extensões concretas, o interpretador renderiza uma interface concretamente.

Cada elemento (*nó*) da árvore possui: uma identificação, a referência de um *widget* concreto e uma lista parâmetros que serão carregados na instanciação de cada *widget*. Conforme dito na seção 3.4.1, os *widgets* concretos devem ser capazes de gerar a sua própria renderização e para a arquitetura apresentada, os *widgets* podem ser implementados utilizando classes de uma linguagem orientada a objetos.

Como ponto de partida para a renderização, o interpretador explora a árvore recebida a partir do nó raiz, utilizando uma busca em profundidade à esquerda (*Depth First Search - DFS*), sendo a ordem de busca ilustrada na Figura 16.

Para cada nó explorado, são capturados os parâmetros e a referência do *widget* concreto, sendo então gerada uma instância do *widget* concreto definido.

Nessa etapa, as extensões concretas também são instanciadas, conforme declarada na lista de extensões concretas.

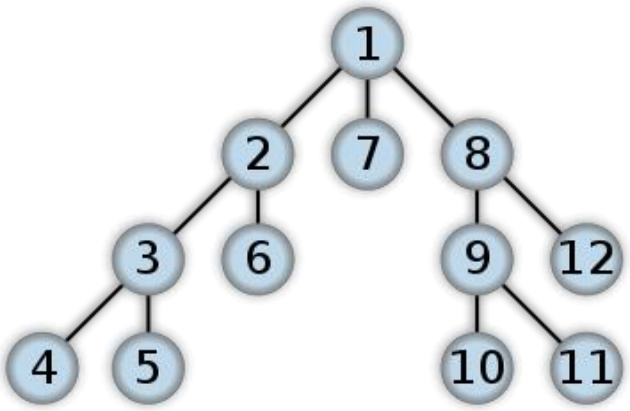

Figura 16 - Exemplo de ordem de exploração dos elementos de uma árvore de composição de interface utilizando uma DFS

Com todos os *widgets* concretos e extensões instanciadas na árvore, pode ser iniciado o processo de renderização recursiva. A recursão é iniciada no nó raiz seguindo a DFS, e quando um nó folha for localizado, o *widget* instanciado se renderiza, retornando o resultado para o nó superior (*pai*). A recursão e conseqüentemente a renderização serão finalizadas quando o nó raiz for renderizado.

Alguns *widgets* possuem dependências de outros *widgets*, como por exemplo, um *widget* de calendário *Javascript* que precisa inserir nos cabeçalhos da página *HTML* a carga de alguma biblioteca ou função. Com isso, um mecanismo de resolução dependências será necessário para que a renderização possa ocorrer sem maiores problemas.

A seção 4.2.8.2 apresenta a implementação de um interpretador de interfaces escrito em linguagem *Ruby*[40], que possui as características mencionadas nesse capítulo.

Para finalizar, após o interpretador concreto concluir a renderização, o módulo de interfaces retorna para a operação solicitante da interface (no modelo comportamental), a interface final renderizada.

[40] http://www.ruby-lang.org

4.2. Arquitetura de implementação

O tópico a seguir apresenta a arquitetura de implementação de um interpretador de modelos descritos segundo o método modelagem de interfaces orientadas por regras, apresentado no capítulo 3, de tal maneira que, a partir de modelos fornecidos pelo projetista, o interpretador é capaz de interpretar e renderizar as interfaces concretamente. Também foi criado um ambiente de autoria dos modelos de interface que será apresentado na seção 4.3.

4.2.1. Componentes chave

No desenvolvimento desse trabalho, os componentes dos módulos foram principalmente escritos em linguagem Ruby, com exceção dos *Widgets* Concretos que, além de Ruby também utilizam outras tecnologias como HTML, JavaScript, Json, CSS e bibliotecas de componentes de terceiros. Esse assunto será melhor explorado na seção 4.2.8.

O ambiente de modelagem de aplicações e a máquina de regras de produção utilizadas serão comentados a seguir, porém outros componentes de software serão comentados conforme sua relevância nos tópicos em que forem citados.

4.2.1.1. Synth

O ambiente para modelagem de aplicações utilizado no desenvolvimento desse trabalho foi o *Synth*, que foi criado utilizando *Ruby on Rails*[41] que é um *framework* MVC (Model–view–controller) para desenvolvimento de aplicações web.

> O Synth é um ambiente de desenvolvimento que dá suporte à construção de aplicações modeladas segundo o método SHDM, fornecendo um conjunto de módulos capazes de receber como entrada os modelos gerados na execução das etapas do método SHDM e produzir como saída uma aplicação hipermídia descrita por estes modelos. O Synth também dispõe de um ambiente de autoria que facilita a inserção e edição destes modelos através de uma interface gráfica de formulários que pode ser executada em qualquer navegador de internet.
>
> [Bomfim, 2011]

[41] http://rubyonrails.org/

O ambiente de interpretação e autoria de interfaces existentes anteriormente no Synth, baseados na proposta de Luna [2009], foram removidos para implementação da arquitetura aqui apresentada.

Para a compreensão aprofundada das etapas de modelagem utilizando o Synth, sugere-se a leitura do capítulo 4 da dissertação de Bomfim [2011].

4.2.1.2. Wongi-Engine

Outro componente chave para o desenvolvimento desta arquitetura foi uma máquina de regras de produção, que nesse trabalho foi a *Wongi-Engine*[42]. Algumas características existentes foram importantes na escolha:

- Ser uma implementação do tradicional algoritmo de Rete baseando-se no trabalho de [Doorenbos, 1995];
- Ter boa legibilidade oferecida na DSL de regras;
- Ter seus fatos serem representados em triplas <*sujeito predicado objeto*>, que é um formato mais próximo do utilizado no Synth, já que seus dados são persistidos como recursos RDF[43], também representados em triplas;
- Por prover um mecanismo facilitado de extensão dos termos (condições) aceitos na DSL das regras;
- Por ser escrito em linguagem Ruby, isso também torna mais natural a captura e utilização de dados disponíveis no ambiente de execução para compô-las. Além disso, é possível utilizar qualquer expressão da linguagem durante a descrição dos fatos e regras.

A seção 4.2.4 apresenta a DSL geral de regras suportada pelo Wongi-Engine, além de algumas extensões criadas para esse trabalho.

[42] https://github.com/ulfurinn/wongi-engine

[43] Resource Description Framework

4.2.2.
Arquitetura em Ruby

A Figura 17 é o diagrama de classes Ruby do módulo de interfaces implementado sobre o Synth, projetado com base na arquitetura conceitual apresentada na Figura 15. As setas simples entre as classes indicam que uma instancia do objeto referenciado será criada, as demais seguem a notação padrão UML (*Unified Modeling Language*).

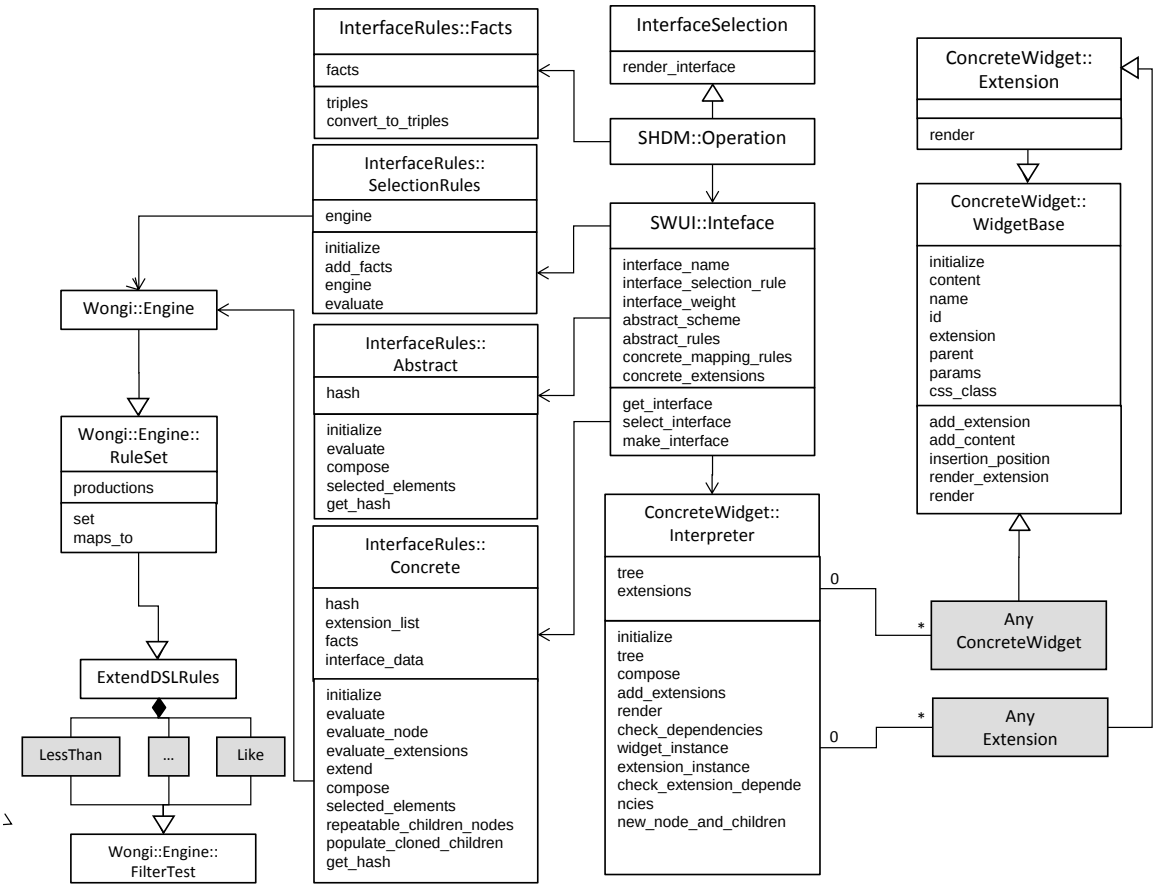

Figura 17 – Diagrama de classes da arquitetura do módulo de interfaces

Uma interface pode ser invocada através de uma operação externa persistida pela classe SHDM::Operation. Na descrição de uma operação, o método render_interface deve receber dois parâmetros como entrada. O primeiro é um conjunto de dados para as regras (fatos), representados como um Hash Ruby (conjunto de pares chave - valor) e o segundo são os dados de mapeamento concreto, também representado com um Hash Ruby.

Em seguida, o método render_interface efetua duas operações:

- Instancia um objeto da classe InterfaceRules::Facts que converte o Hash de fatos das regras para um *Array Ruby*[44] de triplas no formato [‘Sujeito’, ‘Predicado’, ‘Objeto’]. Essas triplas estão no formato aceito pela máquina de regras Wongi-Engine;
- Invoca o método get_interface da classe SWUI::Interface repassando as triplas e os dados de interface para o seletor de interface.

A seção 4.2.3.1 demonstra como descrever esses dados de entrada e provê alguns exemplos.

A troca de mensagem entre as classes pode ser mais bem compreendida através do diagrama de seqüência apresentado na Figura 18, que inicia com a requisição do usuário até o interpretador concreto retornando a interface renderizada.

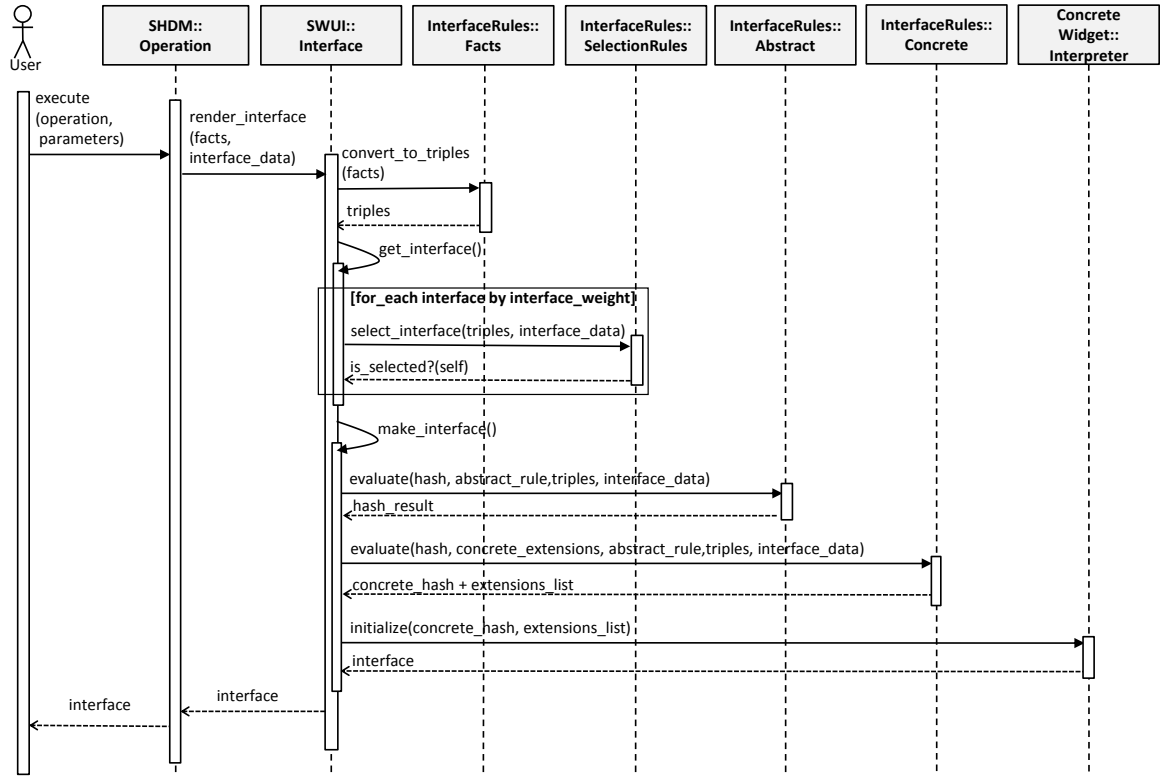

Figura 18 – Diagrama de seqüência ilustrando a troca de mensagem entre as classes

Quando a classe SWUI::Interface recebe as triplas e os dados da interface, começam as etapas de avaliações das regras utilizando instâncias do Wongi-Engine (WE). Em todas as etapas, as triplas de fatos serão inseridas na base de

[44] http://www.ruby-doc.org/core-1.9.3/Array.html

conhecimento do WE e os dados de interface estarão disponíveis como variáveis locais, portando visíveis nas regras. Algumas classes do WE foram estendidas tanto para suportar novos termos na DSL de regras, quanto para inserir novos métodos utilizados na definição das regras de produção. Essa DSL será detalhada na seção 4.2.4.

A primeira avaliação de regras realizada é de seleção de interfaces, que é invocada pelo método select_interface da classe SWUI::Interface. As interfaces dessa classe são ordenadas por peso (propriedade interface_weight). Cada interface tem a sua propriedade interface_selection_rule avaliada por uma instância da classe InterfaceRules::SelectionRules. A primeira interface que na sua avaliação gerar um novo fato no formato *[“rule”, “selected”, true]* será selecionada.

Com uma instância da interface selecionada, o método make_interface é invocado. Ele é responsável por instanciar: o avaliador abstrato, o avaliador concreto e por fim o interpretador concreto.

O avaliador abstrato é implementado pela classe InterfaceRules::Abstract. A composição abstrata é descrita através de um Hash Ruby, sendo armazenada na propriedade abstract_scheme e as regras pela propriedade abstract_rules, ambas da instância da interface. Uma instância do avaliador abstrato é inicializada com o Hash da composição abstrata. O método evaluate recebe as regras, executa a máquina de regras e, como resultado, também são geradas as triplas no formato *[“nome_do_elemento_abstrato”, “selected”, true]* para os elementos que devem ser selecionados. Com isso o método compose percorre o Hash da composição abstrata descartando os elementos que não foram selecionados. As sintaxes utilizadas para declaração da composição abstrata e das regras de seleção são demonstradas na seção 4.2.6.

O próximo passo é feito pelo avaliador concreto, implementado pela classe InterfaceRules::Concrete. Similar ao avaliador abstrato, o avaliador concreto é instanciado com o Hash da composição abstrata resultante. As regras de mapeamento concreto são armazenadas pela propriedade concrete_mapping_rules e concrete_extensions da interface. Diferente do avaliador abstrato, que consegue avaliar todos os elementos da interface de uma vez, o avaliador concreto precisa executar a avaliação das regras (método evaluate_node) para cada elemento da interface, nos casos em que uma coleção de dados é fornecida e então iterada pelo

método compose. Nesses casos, a ramificação do Hash é clonada e tem seus parâmetros populados com os dados do item da coleção (método populate_cloned_children). Após isso, a composição concreta da interface é gerada, sendo essa implementada também como um Hash Ruby compatível com o interpretador concreto.

Com a composição concreta realizada, um Array com as extensões concretas é gerado pelo método add_extention, também no formato compatível com o interpretador concreto.

Como última etapa, o interpretador concreto implementado pela classe ConcreteWidget::Interpreter, é invocado, recebe o Hash da estrutura concreta e o Array de extensões abstratas e então retorna para a operação a interface concreta renderizada.

A arquitetura do interpretador concreto será explorada separadamente na seção 4.2.8.

4.2.3. Base de conhecimento das regras e dados de mapeamento concreto

Conforme visto no tópico anterior, para que a arquitetura de regras implementada possa ser acionada, é necessário informar dois conjuntos de dados: os **dados das regras** e **dados de mapeamento**. Esses dados devem ser fornecidos através de uma operação externa do módulo comportamental do Synth. Uma operação é declarada diretamente em código Ruby. Sendo assim, o método render_interface deve receber, como entrada, objetos do tipo Hash do Ruby. São aceitas as sintaxes de Hash tanto da versão 1.8 quanto da versão 1.9 da linguagem Ruby, conforme exemplo do Quadro 3.

```
#Sintaxe para Ruby 1.8
hash1 = { :key1 => { :a => 'value' }, :key2 => Object.new}

#Sintaxe para Ruby 1.9
hash2 = { key1: { a: 'value' }, key2: Object.new}
```

Quadro 3 – Diferentes sintaxes aceitas para declaração de um Hash Ruby

Sistemas de regras de produção são formados basicamente por suas produções ou regras e por asserções ou fatos. O conjunto desses fatos é chamado

de *memória de trabalho* (WM[45]) e cada fato que a compõe a WM é chamado de *elemento da memória de trabalho* (WME[46]) [GUPTA et al., 1983].

No Wongi-Engine (WE), todos os fatos são representados por triplas no formato <"Sujeito", "Predicado", "Objeto"> [SOKOLOV, 2012]. Assim, um objeto que expresse as propriedades de uma pessoa e suas relações sempre poderá ser decomposto em triplas, conforme exemplo do Quadro 4.

```
<WME "Alice", "age", 35 >
<WME "Alice", "email", "alice@mail.com" >
<WME "Alice", "friend", "Bob" >
```

Quadro 4 – Exemplo de propriedades de 'Alice' em triplas

4.2.3.1. Geração de triplas para base conhecimento

Para que possa ser utilizado como fatos da base de conhecimento do WE, o conjunto de dados de regras precisa ser convertido em triplas. Para isso foi implementado, na classe InterfaceRules::Facts, um conversor capaz de detectar os tipos de dados fornecidos e decompô-los em triplas.

```
#= Objeto representando uma instancia de 'pessoa'
p = FOAF::Person.new("http://my_id")
p.name = "Alice"
p.age = 35

#= Hash com parâmetros de navegação
nav_params = { date_from: "Dec,10, 2012", total: 2 }

#= Hash de fatos
facts = { person: p, params: nav_params, flag: true }

#= Triplas geradas
<WME "person", "uri", "http://id" >
<WME "person", "rdf:about", "http://id" >
<WME "http://id", "name", "Alice" >
<WME "http://id", "age" , 35 >
<WME "http://id", "class", "FOAF::Person" >
<WME "http://id", "rdf:type", "FOAF::Person" >
<WME "params", "date_from", "Dec,10, 2012" >
<WME "params", "total", 2 >
<WME "flag"  , "literal", true >
```

Quadro 5 – Exemplo de Hash de fatos e as triplas geradas a partir delas

Na conversão para triplas, o índice **chave** de cada entrada no Hash ocupará a posição de **sujeito** na relação. As propriedades diretas de objetos de classes do

[45] Working Memory

[46] Working Memory Element

modelo navegacional do Synth ou as chaves de outro Hash serão os **predicados** e seus valores ocuparão a posição de **objetos**. Valores literais aparecerão com um predicado especial chamado **literal**.

O exemplo do Quadro 5 fornece três tipos diferentes de dados para a geração de triplas: O primeiro é um objeto do tipo Pessoa (Foaf::Person), o segundo é o Hash simulando parâmetros de navegação e diretamente uma entrada chamada **flag** com o literal **true**. Para esses dados, o conversor de fatos gerará as triplas indicadas ao final do quadro.

Verificando um pouco melhor o exemplo do objeto do tipo Foaf::Person apresentado no Quadro 5, percebe-se que, mesmo sem estar declaradas, algumas triplas novas foram geradas para **person** com o predicados especiais. Isso ocorre porque o conversor avalia o tipo de cada objeto fornecido e trata de maneira diferente dois casos:

- Quando o objeto for **subclasse da classe RDFS::Resource**[47] ou instâncias de classes do modelo navegacional do SHDM[48]. Para esses casos serão geradas triplas para as *propriedades diretas* existentes no objeto e também triplas indicando:
  - O **tipo** do objeto (predicados "class" e "rdf:type");
  - A identificação (predicados "uri" e "rdf:about").
- Quando a chave indicada no hash for **user_agent** acompanhado do valor da variável de ambiente *HTTP_USER_AGENT*. Nesse caso o conversor extrairá algumas propriedades utilizando uma biblioteca de *parser* específica[49] com os seguintes predicados:
  - **browser**: nome do navegador de internet;
  - **browser_version**: versão do navegador;
  - **platform**: nome do sistema operacional;
  - **mobile**: se a plataforma é portátil ou não.

[47] Todos os objetos que representam recursos no Synth são do tipo RDFS::Resource.

[48] No Synth as classes que representam elementos do modelo navegacional são: SHDM::Context::ContextInstance, SHDM::ContextIndex::ContextIndexInstance e NodeDecorator.

[49] https://github.com/josh/useragent

O tratamento específico realizado para o *user_agent* foi feito na intenção de fornecer para as regras informações sobre a plataforma do usuário, agregando assim mais dados sobre o contexto de uso. As triplas geradas através do *user_agent* serão similares as exemplificadas no Quadro 6.

```
facts = { user_agent: request.env["HTTP_USER_AGENT"] }
#= Triplas geradas
<WME "user_agent", "browser", "Chrome">
<WME "user_agent", "browser_version", "19.0.1084.56">
<WME "user_agent", "platform", "Windows">
<WME "user_agent", "mobile", false>
```

Quadro 6 – Exemplo de triplas geradas a partir do user_agent

Como estamos tratando de aplicações modeladas no Synth, certas instâncias de objetos que representam elementos do modelo navegacional também podem ser desejadas como fatos da base de conhecimento, como por exemplo, um índice de um contexto ou um recurso em um contexto. O Quadro 8 apresenta um exemplo completo de código de operação extraído do Synth para tratar a requisição de um nó (recurso) em um contexto.

4.2.3.2. Dados de mapeamento

Conforme já dito, os dados de mapeamento são utilizados nos parâmetros de mapeamento concreto da interface e também podem ser inseridos na composição das regras. Isso é possível porque o índice **chave** de cada entrada no Hash será integrado ao escopo do ambiente de avaliação das regras e também dos parâmetros de mapeamento, sendo acessível como variável local.

```
idx = SHDM::Index.find(index_id)
index = idx.new( params )

interface_data = { index: index, params: params, my_var: "ABC" }
```

Quadro 7 – Exemplo de dados para mapeamento concreto para índice de contexto

O Quadro 8 apresenta um exemplo completo de uma operação externa utilizada no Synth.

```
context_id   = params.delete(:id)
node_id      = params.delete(:node)
context_i    = SHDM::Context.find(context_id)
context      = context_i.new(params)
```

```
current_node = NodeDecorator.new(RDFS::Resource.new(node_id),
context)

#== Facts
facts = {
  navigational_element: context,
  current_node: current_node,
  user_agent: request.env["HTTP_USER_AGENT"],
  params: params
}
#== Interface data for mapping
interface_data = {
  context: context,
  current_node: current_node
}
interface = render_interface(facts, interface_data)
render :text => interface
```

Quadro 8 – Exemplo de código Ruby de uma operação de contexto do SHDM

O código da operação se inicia com a captura dos parâmetros de navegação, armazenando a identificação do contexto (*context_id*) e do nó navegacional (*node_id*). Esses valores são utilizados na instanciação dos objetos do modelo navegacional (*context_i* e *context*) e do nó corrente decorado pelas propriedades do contexto (*current_node*).

Na sequência são declarados os dados para os fatos (*facts*), onde serão convertidas para triplas as propriedades do contexto atual (navigational_element), o nó navegacional corrente (*current_node*), as informações da plataforma do usuário (*user_agent*) e os parâmetros de navegação (*params*). Como dados para mapeamento concreto (*interface_data*), também são enviados o contexto (context) e nó navegacional corrente (current_node)

A função render_interface recebe os dados das regras (*facts*) e de mapeamento (*interface_data*) e retorna a *String* da interface gerada para ser renderizada pela função render padrão.

4.2.4. DSL geral de regras

Esse tópico apresenta como descrever regras através da DSL original do Wongi-Engine e também condições estendidas que foram desenvolvidas para esse trabalho.

De maneira genérica, a declaração de uma regra no Wongi-Engine se dá através de blocos de condições agrupadas por seções. Originalmente uma seção de

condições é definida pelo termo *forall* e para execução de alguma ação, caso as regras sejam correspondidas, o bloco make deve ser declarado [SOKOLOV, 2012]. Todavia, nesse trabalho, esses blocos foram absorvidos pelas expressões *set* e *maps_to* a fim de facilitar a escrita das regras, já que os avaliadores implementados tem em seu código a definição das ações a serem tomadas. Essas expressões serão exploradas nos tópicos adiante.

Para escrever uma regra simples, pode-se fazer uso de um mecanismo chamado de *token*. Um *token* pode ser utilizado como um tipo de variável para compor múltiplas condições. Um *token* deve ser nomeado sempre iniciado por dois pontos e letra maiúscula, como por exemplo ***':TokenA'***. Outro tipo de *token* é o **':_'**, que vale como um caractere coringa, aceitando como válido qualquer valor aonde for inserido.

```
#= Fatos
<WME "person", "uri", "http://id" >
<WME "person", "rdf:about", "http://id" >
<WME "http://id", "name", "Alice" >
<WME "http://id", "age" , 35 >
<WME "http://id", "class", "FOAF::Person" >
```

Quadro 9 – Exemplo de triplas de fatos

Seguindo o mesmo raciocínio do formato de tripla <Sujeito, Predicado, Objeto> e supondo-se que existam, na base de conhecimento, os fatos descritos no exemplo do Quadro 9, consideremos uma regra que seja válida no seguinte caso: "valide caso exista alguma tripla que tenha o Predicado **name** e o Objeto **Alice**". Uma declaração compatível com esse caso, utilizando a DSL de regras e os *tokens*, poderia ser feita conforme o Quadro 10.

```
set{
  has :_, "name", "Alice"
}
```

Quadro 10 – Exemplo básico de regra

Porém, fazendo uso dos *tokens* consegue-se uma composição mais interessante. No exemplo do Quadro 11, a regra só é válida caso todas as condições forem válidas (conjunção), ou seja, caso exista uma pessoa que possua as propriedades *name* e *age*, quaisquer que sejam seus valores.

```
set{
  has "person",    "uri",   :PersonURI
  has :PersonURI, "name",  :_
  has :PersonURI, "age",   :_
}
```

Quadro 11 – Regra para caso exista uma mesma pessoa com a propriedade *name* e *age*

Logo abaixo é exibido outro exemplo para os *tokens*. Além disso, mais uma condição (função) foi utilizada, o *equal* ou *eq*. Nesse caso a regra só é válida caso exista uma pessoa que tenha a idade igual a 18.

```
set{
  has "person",   "uri",  :PersonURI
  has :PersonURI, "age",  :Age
  equal :Age, 18
}
```

Quadro 12 – Regra válida para uma pessoa com idade igual a 18

Qualquer valor literal ou objeto Ruby é válido na composição das condições das regras, podendo-se utilizar, por exemplo, a data e hora corrente (obtida do computador) ou ainda as variáveis dos dados de mapeamento concreto. O Quadro 13 demonstra uma condição aonde à regra só é válida para o mês de fevereiro.

```
set{
  equal Time.now.month, 2
}
```

Quadro 13 – Regra válida somente quando o mês corrente for fevereiro

Segue abaixo uma lista com todas as condições (nativas ou estendidas) aceitas, além da condição *has*, para descrever as regras no Wongi-Engine:

- *neg* {}- Exatamente o inverso da condição *has*, portanto só será aceita se nenhum ocorrência existir;
  Apelido(alias): *missing*
  Formato: *neg subject, predicate, object*
- *maybe* {} – É uma condição válida para uma ocorrência que exista ou não, sendo mais útil para criar *tokens* para outras condições;
  Apelido(alias): *optional*
  Formato: *maybe subject, predicate, object*
- *none* {} – Funciona como um bloco que agrupa outras condições e só será válida se não existirem ocorrências para todo o bloco. Corresponde a uma expressão de negação. not (a and b and ..)
- *any* { } – Funciona como disjunção, agrupando outras condições.
  Formato: any { option { ... } ... }
- *same* – Valida a condição caso os argumentos sejam iguais.
  Apelido(alias): *equal, eq*
  Formato: *same x, y*

- *diff* – Valida a condição caso os argumentos sejam diferentes.
  Apelido(alias): *ne*
  Formato: *diff x, y*
- *assert* {} – Valida a condição caso o bloco retorne verdadeiro. É útil para construir condições não suportadas pelas condições anteriores. A lista de todos os *tokens* é enviada como parâmetro.
  Formato: *assert { |tokens| ... }*
  Exemplo: assert {| t | t[:Age] >=18 }
- *greaterThan* – Valida a condição caso o primeiro argumento seja maior que o segundo.
  Apelido(alias): *gt*
  Formato: *gt x, y*
- *lessThan* – Valida a condição caso o primeiro argumento seja menor que o segundo.
  Apelido(alias): l*t*
  Formato: *lt x, y*
- *greaterThanEqual* – Valida a condição caso o primeiro argumento seja maior ou igual ao segundo.
  Apelido(alias): *gte*
  Formato: *gte x, y*
- *lessThanEqual* – Valida a condição caso o primeiro argumento seja menor ou igual ao segundo.
  Apelido(alias): *lte*
  Formato: *lte x, y*
- *like* – Valida a condição caso o segundo argumento esteja contido na string do primeiro (sub-string).
  Apelido(alias): *as*
  Formato: *like x, y*
  Exemplo: like “palavra”, “pala”

4.2.5. Seleção de interfaces

A primeira etapa de avaliação de regras na arquitetura de interface é a seleção de interfaces. Para isso, cada interface deve possuir regras de seleção que expressem as condições em que ela pode ser acionada.

Como a arquitetura do módulo de interfaces foi implementada sobre o Synth, suportando assim os modelos do SHDM, alguns casos comuns de ativação de interface podem ser deduzidos. Por exemplo, uma interface pode ser projetada para:

- Exibir qualquer índice de um contexto;
- Exibir um índice de um contexto específico;
- Exibir qualquer recurso (nó) em um contexto;
- Exibir um recurso (nó) em contexto específico;
- A interface de uma operação específica.

A partir disso pode-se imaginar muitos outros casos que combinem essas situações adicionadas de outras condições, como algumas situações apontadas na seção 3.2.

Para servir como critério de desempate no caso de conflito entre as regras, a propriedade *interface_weight* precisa receber um valor numérico indicando qual é o peso da interface, sendo o de menor valor o mais prioritário.

A descrição de uma regra de seleção de interface deve utilizar a expressão *set* e um bloco de condições descritas no tópico anterior.

```
<WME "user_agent" "browser" "Firefox">
<WME "user_agent" "platform" "Windows NT 6.1">
<WME "user_agent" "mobile" false>
<WME "params" "controller" "execute">
<WME "params" "action" "index">
<WME "nav_element" "uri" "http://base#1234">
<WME "http://base#1234" "class" "RDFS::Resource">
<WME "http://base#1234" "class" "SHDM::ContextIndex">
<WME "http://base#1234" "index_title" "My Idx Default">
<WME "http://base#1234" "index_name" "DefaultIDX">
```

Quadro 14 – Fatos para índice de contexto

Para ilustrar algumas situações, suponha que os fatos do Quadro 14 foram gerados na requisição de uma operação externa. Analisando os fatos, basicamente os dados de regras fornecidos foram: os parâmetros de navegação (*params*), um índice de contexto como elemento navegacional (*nav_element*) e os dados do user_agent.

Para criar uma regra genérica o suficiente para que corresponda a qualquer índice de contexto, a declaração seria como a do Quadro 15.

```
set{
  has "nav_element",   "uri",  :URI
  has :URI, "class", "SHDM::ContextIndex"
}
```

Quadro 15 – Regra para selecionar qualquer índice de contexto

A interface será selecionada quando uma operação fornecer um elemento navegacional (nav_element) da classe SHDM::ContextIndex.

Para restringir a seleção de uma interface para um índice de contexto específico, como por exemplo, o de nome *DefaultIDX*, basta adicionar mais uma condição conforme o Quadro 16.

```
set{
  has "nav_element",   "uri",  :URI
  has :URI, "class", "SHDM::ContextIndex"
  has :URI, "index_name", "DefaultIDX"
}
```

Quadro 16 – Regra para selecionar uma interface para o índice de nome DefaultIDX

Supondo outra situação, caso a interface seja válida somente para índices de contextos em computadores de qualquer versão da plataforma *MS Windows*. As condições seriam como a do Quadro 17.

```
set{
  has  "nav_element", "uri",  :URI
  has  :URI, "class", "SHDM::ContextIndex"
  has  "user_agent",  "platform",  :Platform
  like :Platform, "windows"
}
```

Quadro 17 – A interface só será válida para plataforma MS Windows

Diversas outras situações poderiam servir como exemplo, mas conclui-se que o mecanismo de regras de seleção é flexível o suficiente para o projetista definir com bom grau de liberdade se uma interface atenderá um ou diversos contextos de uso.

4.2.6. Composição abstrata

Conforme o método para modelagem de interfaces proposto nesse trabalho, a modelagem abstrata representa o nível mais básico dos elementos perceptíveis da interação entre o usuário e a aplicação. Esses elementos são representados

através dos *widgets* abstratos organizados hierarquicamente em uma arquitetura de árvore.

O que será demonstrado a seguir é uma notação para descrição da composição abstrata e também da declaração de regras de seleção de elementos abstratos.

4.2.6.1. Sintaxe de declaração da interface abstrata

A descrição da composição abstrata implementada na arquitetura desse trabalho, utiliza uma representação em Hash Ruby versão 1.9. Apesar de compatível com a sintaxe da versão do Ruby 1.8, recomenda-se o uso da versão mais recente por se mostrar mais concisa.

Os *widgets* abstratos aceitos são: *AbstractInterface*, *CompositeInterfaceElement*, *ElementExhibitor*, *IndefiniteVariable*, *PredefinedVariable* e *SimpleActivator*.

As propriedades descritas no método foram implementadas como chaves de um Hash, sendo aceitos como propriedades os seguintes termos:

- **name** – identificação do elemento abstrato. Deve ser único para toda composição e é uma propriedade obrigatória;
- **widget_type** – nome do tipo do *widget* abstrato utilizado no elemento em questão e é uma propriedade obrigatória;
- **repeatable** – Indica se os elementos filhos são repetíveis, representando uma coleção. Aceita os valores *true* ou *false* e não é uma propriedade obrigatória;
- **children** – recebe uma lista (Array) dos elementos *filhos* do nó em questão, que se trata de uma composição abstrata com a mesma estrutura. Não é uma propriedade obrigatória.

```
{name: "main_page", widget_type: "AbstractInterface",
 children:[
  {name: "header", widget_type: "CompositeInterfaceElement",
   children:[
    {name: "logo", widget_type: "SimpleActivator"},
    {name: "title", widget_type: "ElementExhibitor"}
   ]},
  {name: "items", widget_type: "CompositeInterfaceElement",
   repeatable: true, children:[
```

```
    {name: "link", widget_type: "SimpleActivator"}
    {name: "description", widget_type: "ElementExhibitor"}
  ]}
```

Quadro 18 – Exemplo de composição abstrata em Hash Ruby

O Quadro 18 apresenta um pequeno exemplo de composição abstrata utilizando todas as propriedades citadas anteriormente. O exemplo basicamente ilustra uma interface (*main_page*) com um cabeçalho (*header*) que possui uma logomarca (*logo*) e um título (*title*). Após, uma coleção de itens onde cada um possui uma ancora de navegação (*link*) e uma descrição (*description*).

O elemento da raiz é representado por um *widget* abstrato do tipo *AbstractInterface* e seus elementos internos (*filhos*) estão num Array na propriedade *children*. Outro ponto interessante é o uso da propriedade *repeatable* para o elemento *items*.

Nenhuma propriedade adicional precisa ser incluída na modelagem abstrata, ignorando nesse momento os dados que irão renderizar a composição, mantendo somente a hierarquia dos itens que compõem a interface.

4.2.6.2. Descrição de regras de seleção de elementos abstratos

Com a composição abstrata descrita através da sintaxe em Hash Ruby vista no tópico anterior, o projetista de interfaces pode definir regras para condicionar a inclusão dos elementos em função dos fatos fornecidos.

Conforme as premissas estabelecidas na seção 3.3.2, caso um elemento não possua regras, ele sempre participará da composição, caso contrário, as condições descritas na regra limitarão a sua inclusão.

A sintaxe utilizada para definir uma regra é praticamente a mesma da seleção de interfaces, utilizando as mesmas condições descritas na DSL. Para a regra de seleção de elementos a expressão *set* precisa do nome informado na propriedade *name* da composição e de um bloco de condições.

O Quadro 19 fornece um exemplo de um elemento com o nome *banner*, que só será inserido na composição caso o índice do contexto tenha o nome *NewsIdx*.

```
set "banner" do
  has "nav_element",   "uri",  :URI
  has :URI, "index_name", "NewsIdx"
end
```

Quadro 19 – Exemplo de elemento selecionado caso o índice for NewsIdx

Um mesmo elemento da composição pode receber mais de uma regra declarada, o que equivale a realizar uma disjunção entre as condições (uma regra ou outra regra, etc).

Para ilustrar um caso de uso, suponha uma composição de interface para exibir informações de um hotel. Devem ser exibidos o nome, cidade e endereço do hotel. Essa composição também deve exibir as tarifas em uma determinada data de *check-in* e *check-out*. Contudo, caso as datas não forem fornecidas, no lugar das tarifas deve ser exibido um formulário com campos para o usuário fornecer as datas de check-in e check-out.

O Quadro 20 contém um exemplo de composição abstrata de interface que pode cumprir com os requisitos informados no caso da interface do hotel citado anteriormente.

```
{name: "main_page", widget_type: "AbstractInterface", children:[
  {name: "header", widget_type: "CompositeInterfaceElement",
   children:[
    {name: "logo", widget_type: "SimpleActivator"},
    {name: "title", widget_type: "ElementExhibitor"}
   ]},
  {name: "content", widget_type: "CompositeInterfaceElement", children:[
    {name: "hotel_name", widget_type: "ElementExhibitor" },
    {name: "hotel_city", widget_type: "ElementExhibitor" },
    {name: "hotel_address", widget_type: "ElementExhibitor" },
    {name: "hotel_prices", widget_type: "CompositeInterfaceElement",
     children:[
      {name: "checkin", widget_type: "ElementExhibitor" },
      {name: "checkout", widget_type: "ElementExhibitor" },
      {name: "prices", widget_type: "CompositeInterfaceElement",
       repeatable: true, children:[
        {name: "price", widget_type: "ElementExhibitor"}
      ]}
    ]},
    {name: "search_rates", widget_type: "CompositeInterfaceElement",
     children:[
      {name: "title_search", widget_type: "ElementExhibitor" },
      {name: "checkin_field", widget_type: "IndefiniteVariable"},
      {name: "checkout_field", widget_type: "IndefiniteVariable"},
      {name: "send", widget_type: "ElementExhibitor"}
    ]},
  ]},
]}
```

Quadro 20 – Interface de exibição de um hotel e suas tarifas

Analisando o Quadro 20 percebe-se que o elemento *hotel_prices* possui elementos (filhos) para exibir as tarifas do hotel e o elemento *search_rates* os campos do formulário para informar as datas de check-in e check-out.

Para condicionar a inclusão de um trecho da composição ou de outro poderia se definir uma regra para cada um deles, conforme o exemplo do Quadro 21.

```
set "hotel_prices" do
  has "params", "checkin_field", :_
  has "params", "checkout_field", :_
end

set "search_rates" do
  neg "params", "checkin_field", :_
  neg "params", "checkout_field", :_
end
```

Quadro 21 – Exemplo de regra de seleção de elementos abstratos

O exemplo do Quadro 21 demonstra uma regra para o elemento *hotel_prices* que só será válida caso a lista de parâmetros de navegação possua valores de *checkin_field* e *checkout_field*. Para o elemento *search_rates* a condição é exatamente inversa bastando utilizar a condição *neg* para expressar isso.

4.2.7. Declaração de regras de mapeamento concreto

A última etapa de avaliação de regras do método é a de mapeamento concreto, onde são definidos quais *widgets* concretos deverão renderizar os elementos da interface. Na arquitetura implementada, cada elemento da composição abstrata deverá possuir um regra que defina qual *widget* concreto será utilizado e que parâmetros serão enviados para o *widget* concreto escolhido.

Esse tópico aborda de forma genérica a declaração das regras e parâmetros de renderização para os *widgets* concretos. Todavia, detalhes da arquitetura de construção dos *widgets* concretos podem ser vistos na seção 4.2.8.

Diferente das regras de seleção abstrata, o mapeamento concreto sempre deverá ser definido para todos os elementos abstratos. Sendo assim, os elementos abstratos sem mapeamento não serão renderizados.

Para realizar o mapeamento, as regras e os parâmetros descritos utilizam os valores informados na operação externa através do hash de dados de mapeamento concreto. Esses dados estarão visíveis como variáveis locais ao escopo da interface, ficando disponíveis através do nome utilizado na *chave* do Hash criado.

A expressão utilizada para declarar uma regra de mapeamento é o *maps_to* seguido de alguns parâmetros:

- **abstract** – recebe a identificação do elemento abstrato, a mesma declarada na propriedade *name* da composição abstrata. É uma propriedade obrigatória;
- **concrete_widget** – recebe o nome do *widget* concreto desejado para a renderização. É uma propriedade obrigatória;
- **params** – é declarada por um Hash Ruby com valores a serem repassados para a renderização do *widget* concreto. É uma propriedade opcional para alguns *widgets*, sendo necessário verificar os parâmetros requeridos no *widget* utilizado (definido no arquivo de manifesto[50]);
- **bloco de regras (do end)** – bloco de condições que definem quando o mapeamento será realizado. A declaração desse bloco é opcional, caso não seja definido o mapeamento sempre será realizado.

```
maps_to abstract: "checkin_dt", concrete_widget: "JQueryDatePickerInput"

maps_to abstract: "main_page", concrete_widget: "HTMLPage",
params: { title: index.index_title, css_class: "page" }

maps_to abstract: "attribute_label", concrete_widget: "HTMLLabel",
params: {content: entry.label, css_class: "common" } do
  diff index.entries.first, entry
end
```

Quadro 22 – Exemplos de regras de mapeamento concreto

O Quadro 22 apresenta três exemplos de regras de mapeamento concreto. Na primeira, o elemento *checkin_dt* será mapeado para o *widget* concreto *JQueryDatePickerInput* sem o uso de nenhum parâmetro e nem condições para restringir a sua renderização. Na segunda, o elemento *main_page* informa os parâmetros *title* e *css_class* para renderização do *widget HTMLPage*. A última regra utiliza, além dos parâmetros, uma condição para que a renderização somente ocorra caso a variável *entry* não seja a primeira da lista do índice *index.entries*.

Nessa etapa é de grande importância que o projetista tenha um bom conhecimento de como manipular os dados de mapeamento informados na operação externa, pois isso é essencial para fornecer as informações corretas nos parâmetros de renderização. Além disso, esses dados provavelmente também

[50] Os arquivos de manifesto dos *widgets* concretos serão abordados na seção 4.2.8.1.5 .

serão utilizados nas regras de mapeamento concreto. Algum conhecimento da linguagem Ruby também auxiliará para um melhor aproveitamento na descrição das regras, já que essa etapa é a mais próxima possível da renderização concreta a ser realizada.

Informações adicionais à renderização, como imagens e páginas de estilo CSS devem ser armazenadas locais visíveis pelo ambiente de execução da aplicação (no caso do Synth podem ser armazenadas no diretório *public*), sendo utilizado nos parâmetros de mapeamento somente o caminho (*path*) para esses arquivos.

4.2.7.1. Coleções de dados

Quando uma composição abstrata de interface utiliza a propriedade *repeatable: true*, o projetista está indicando que uma coleção de dados precisa ser percorrida para fornecer os valores para os seus elementos internos, ou seja, seus filhos. Nesses casos, dois parâmetros especiais devem ser utilizados na regra de mapeamento concreto:

- **collection** – recebe como entrada uma lista de dados, no caso, um Array de objetos Ruby;
- **as** – serve para definir qual o nome da variável de acesso de cada instância ou item da coleção. Essa variável estará no escopo de cada instância para os elementos filhos do elemento que recebeu a coleção. Caso não seja definido parâmetro **as**, a variável no escopo da regra será acessível pelo nome padrão *value*.

```
{name: "attributes", widget_type: "CompositeInterfaceElement",
repeatable: true, children:[
  {name: "attribute_label", widget_type: "ElementExhibitor"},
  {name: "attribute_values", widget_type: "SimpleActivator"}
]}
```

Quadro 23 – Trecho de composição abstrata para ilustrar coleções

Como exemplo do uso de coleções, consideremos o trecho de composição abstrata do Quadro 23. O elemento *attributes* possui a propriedade *repeatable: true* e tem como filhos os elementos *attribute_label* e *attribute_values*.

```
maps_to abstract: "attributes", concrete_widget: "HTMLComposition",
params: { collection: entry.attributes, as: "attribute"  }

maps_to abstract: "attribute_label", concrete_widget: "HTMLLabel",
params: { content: attribute.label  }

maps_to abstract: "attribute_values", concrete_widget: "HTMLAnchor",
params: { content: attribute.name, url: attribute.target_url, css_class:
"highlight"  } do
  eq entry.attributes.first, attribute
end

maps_to abstract: "attribute_values", concrete_widget: "HTMLAnchor",
params: { content: attribute.name, url: attribute.target_url }
```

Quadro 24 – Regras de mapeamento para coleções de dados

Para renderizar a composição indicada no Quadro 23, a regra de mapeamento para o elemento *attributes* deverá utilizar o parâmetro *collection* para receber um Array com a coleção a ser percorrida e o parâmetro *as* para indicar o nome da variável visível no escopo dos elementos filhos.

O Quadro 24 ilustra um conjunto de regras possíveis para renderizar a interface do Quadro 23. O elemento *attributes* recebe no parâmetro *collection* a coleção *entry.attributes* e declara como *attribute* o nome da variável visível com os dados de data item. Para o elemento *attribute_values* foram declaradas duas regras. A primeira incluirá o parâmetro *css_class:"highlight"* para a renderização somente para o primeiro elemento da coleção e os demais casos serão mapeados com a última regra.

Uma interface também pode conter coleções aninhadas em sua composição. Para isso, a composição abstrata também deve ter em seus elementos filhos a propriedade *repeatable:true* definida. Nesses casos, a variável de uma coleção mais externa pode servir de entrada para coleções mais internas.

4.2.7.2. Declarando extensões

Cada elemento que compõe a interface pode receber extensões para seus *widgets* concretos, seja para dar-lhes novas capacidades ou novos comportamentos ou para inserir elementos novos na composição, necessários para complementar a renderização concreta.

Na arquitetura implementada, as extensões são codificadas exatamente da mesma maneira que os *widgets* concretos, sendo uma subclasse da classe dos *widgets* concretos. Inclusive, um *widget* concreto pode ser utilizado como uma extensão, porém uma extensão não pode ser utilizada como *widget* concreto.

A declaração de uma extensão é feita pela expressão *extend*, acrescida de algumas propriedades:

- **nodes** – array com a identificação dos elementos a serem estendidos. Os elementos do array devem possuir o mesmo nome fornecido na propriedade *name* da composição abstrata. Essa propriedade é obrigatória;
- **extension** – nome da extensão ou *widget* concreto que será utilizado. Essa propriedade é obrigatória;
- **params** – hash com parâmetros utilizados na renderização da extensão, funcionando exatamente como os parâmetros do *widgets* concretos. Essa propriedade é opcional;
    - um parâmetros especial é o **insertion_position**, que aceita os valores *after*, *before* e *around*. Se nenhum for definido o valor *after* será utilizado, ou seja, a extensão será incluída após cada elemento indicado na propriedade *nodes*.

```
extend nodes: ['attribute_values'], extension: 'HTMLLineBreak'

extend nodes: ['depart_date'], extension: 'JQueryCopyTo',
params: { target: 'return_date'}

extend nodes: ['leg_go_arrive', 'leg_back_depart'], extension:
'HTMLText', params: { content: " on ", insertion_position: "before"}
```

Quadro 25 – Exemplo de extensões de interface

O Quadro 25 possui alguns exemplos de inclusão de extensões na interface. O primeiro utiliza o *widget HTMLLineBreak* para inserir uma simples quebra de linha após o elemento *attribute_values*. O segundo exemplo utiliza a extensão *JQueryCopyTo* que copia o valor inserido pelo usuário para o elemento *return_date*. O último exemplo mostra um caso interessante, pois estende dois elementos simultaneamente e utiliza o parâmetro *insertion_position* para que sejam inseridos na interface antes dos elementos indicados na propriedade *nodes*.

O apêndice dessa dissertação contém uma lista de todos os *widgets* concretos e extensões desenvolvidas e disponíveis até o momento.

4.2.8. Arquitetura dos Widgets Concretos

Essa seção apresenta a arquitetura de construção dos *widgets* concretos e extensões e também de sua máquina de interpretação.

Conforme visto na seção 3.4.1, os *widgets* concretos são artefatos de software que, ao receber parâmetros de entrada, devem ser capazes de: resolver as dependências com outros *widgets*, adicionar as suas extensões e renderizar-se. Portanto a complexidade da implementação deve ser ao máximo absorvida pelo *widget*, livrando de quem o usa dos detalhes da sua codificação.

Sendo tratados dessa maneira, a demanda por novas funcionalidades ou uso de tecnologias mais recentes sempre poderá ser atendida pela criação de novos *widgets* que poderão ser distribuídos como bibliotecas de software.

Apesar de estar incluído na arquitetura desenvolvida nesse trabalho, o conjunto de *widgets* concretos, extensões e o interpretador podem ser utilizados ou acoplados livremente a outros ambientes que necessitem de geração de interfaces. Isso é possível porque foram desenvolvidos com um esquema aberto de descrição da interface, que será apresentada na seção 4.2.8.2.

4.2.8.1. Estrutura básica de um Widget Concreto

Na arquitetura desenvolvida, um *widget* concreto é construído basicamente da herança da superclasse WidgetBase, de um template Tilt[51] e do seu arquivo de manifesto, todos escritos em linguagem Ruby. Além disso, bibliotecas Javascript, CSS, HTML, XML e qualquer outra tecnologia utilizada para construir aplicações web podem ser acopladas na sua construção.

[51] Templates Tilt serão comentados na seção 4.2.8.1.2.

4.2.8.1.1. Super classes WidgetBase e Extension

A construção de um *widget* concreto inicia-se através da herança da superclasse ConcreteWidget::WidgetBase.

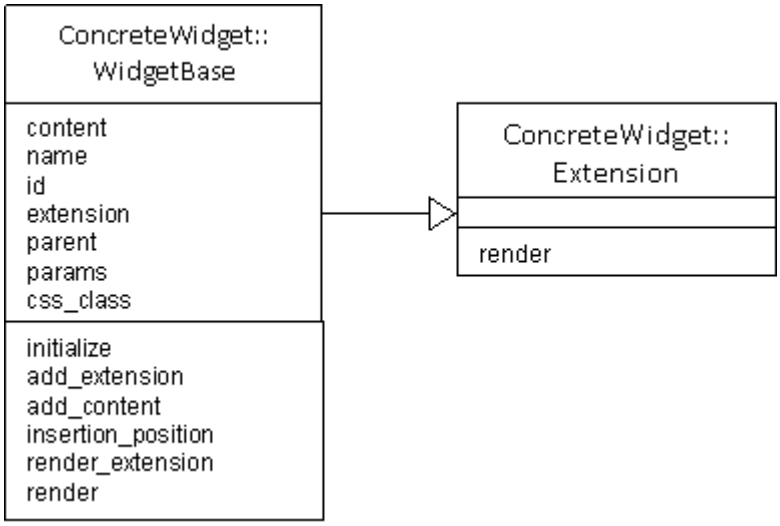

Figura 19 – A classe Widget Base é a superclasse para construção dos *widgets* concretos

Um *widget* pode simplesmente herdar as características dessa classe e somente aplicar um template diferente, como também pode redefinir os métodos e atributos herdados conforme a necessidade.

Os principais atributos da classe WidgetBase são:

- **content** – atributo padrão para recebimento de algum dado para renderização. Atributo obrigatório em todas as instâncias dos *widgets* mesmo sobrescrevendo o construtor (initialize);
- **name** – atributo que identifica a instância do *widget*. Atributo obrigatório;
- **id** – utilizado caso o elemento precise de uma identicação na interface renderizada, como por exemplo em páginas HTML ou instâncias de componentes Javascript. Caso não informado, será utilizado o mesmo do atributo *name*;
- **params** – recebe o hash com os parâmetros enviados para o construtor da classe;
- **css_class** – utilizado em *widgets* que possam receber uma classe CSS;
- **extensions** – Recebe um array com as extensões a serem aplicadas;

Os principais métodos existentes nessa classe são (além dos métodos de acesso aos atributos):

- **initialize** – método construtor da classe, recebe os parâmetros de renderização e atribui para as variáveis de instância.
- **add_content** – utilizado principalmente para receber o conteúdo de renderização das extensões;
- **add_extension** – inclui uma instância de extensão no array de extensões;
- **render_extension** – efetua a renderização de todas as extensões atreladas ao *widget* e insere o resultado conforme informado no parâmetro *insertion_position*.
- **render** – executa a renderização das extensões, verifica se o *widget* possui um template Tilt, executa a renderização do template e retorna o resultado para o interpretador.

Uma extensão herda todos os métodos da classe WidgetBase com exceção do método render que possui um tratamento diferente para a pasta de extensões.

4.2.8.1.2. Templates Tilt

Tilt[52] é uma biblioteca Ruby que funciona como uma máquina de *templates*, dando suporte a diferentes mecanismos de templates existentes, porém com o uso genérico para todas elas. São suportados diversos formatos: Erubis (erb), Markaby (mab), string interpolado, Markdown e muitos outros.

A escolha do Tilt como máquina de templates foi exatamente para dar liberdade para o desenvolvedor de *widgets* concretos de utilizar o formato que esteja mais familiarizado ou que traga maior flexibilidade.

[52] https://github.com/rtomayko/tilt

O template criado é acionado pelo método render da classe WidgetBase, que instancia automaticamente o arquivo de template, ficando nele visíveis todas variáveis de instância criadas e os métodos públicos do widget.

Um template pode ser simples como, por exemplo, o template do *widget HTMLAnchor*, que utiliza template *Erubis*, conforme o Quadro 26.

```
<a class="<%= @css_class %>" id="<%= @id %>" title="<%= @params[:title]
%>" href="<%= @params[:url] %>" /> <%= @content %> </a>
```

Quadro 26 – Template Erubis do *widget* HTMLAnchor

Quando o *widget* HTMLAnchor é renderizado o resultado será similar ao da Figura 20, gerando uma âncora HTML (link).

AOL Research

AT&T Labs - Research, AT&T Labs Research

Academia Sinica

Figura 20 – HTMLAnchor renderizado

Templates também podem ser mais sofisticados, como é o caso do utilizado no *widget* JQueryAjaxAutocomplete, conforme o Quadro 27 que utiliza HTML e Javascript em sua construção.

```
<input type="text" id="<%=@id%>" name="<%=@name%>" class="<%=
@css_class%>" autocomplete="off"/>

<script type="text/javascript" charset="utf-8">
$(document).ready(function() {
 $( "#<%=@id%>" ).autocomplete({
  source: function( request, response ) {
      $.ajax({
          url: "<%= @json_source_url %>",
          dataType: "<%= @data_type %>",
          data: {
             <%= @search_parameter %>: request.term
          },
          success: function( data ) {
            response(
              $.map( data<%= @node_json_result_element %>,
              function( item ) {
                return  <%= @hash_result_format %>
              }
            ));
          }
      });
  },
```

```
  minLength: 2
});
});
</script>
```

Quadro 27 – Template do *widget* JQueryAjaxAutocomplete

O resultado da renderização JQueryAjaxAutocomplete é exemplificado na Figura 21, que exibe uma caixa de texto com a opção de auto-completar.

Figura 21 – Template JQueryAjaxAutocomplete renderizado

4.2.8.1.3. Dependências entre *widgets* concretos

Alguns *widgets* concretos possuem como dependência a existência de outro *widget* no esquema de composição da interface, para que seja possível a sua renderização. Por exemplo, o *widget* JQueryAjaxAutocomplete utiliza a biblioteca JQuery[53] em sua construção. Porém, para a mesma poder ser utilizada, é exigida a carga dos arquivos no cabeçalho de uma página HTML.

Para resolver essas questões, um mecanismo de busca por dependências foi criado, onde o *widget* dependente declara quais os *widgets* de que depende e recebe em outro método a referencia para a instância do *widget* dependido.

O interpretador efetua uma busca na árvore da composição recebida procurando um *widget* da mesma classe informada pelo método *dependencies* do *widget*. Ao encontrar essa instância, o método *run_dependencias* tem acesso a todos os métodos públicos da instância recebida, podendo assim efetuar as manipulações necessárias.

```
def dependencies
  ["HTMLPage"]
end
```

[53] http://jquery.com/

```
def run_dependencies(obj)
  obj.include_js(["/_shared/js/jquery-1.7.2.min.js",
                  "/_shared/js/jquery-ui-1.8.21.custom.min.js"])
  obj.include_css(["/_shared/css/ui-lightness/jquery-ui-1.8.21.css"])
end
```

Quadro 28 – Dependências do *widget* JQueryAjaxAutocomplete

O Quadro 28 exemplifica a resolução das dependências para o *widget* JQueryAjaxAutocomplete, informando que depende do *widget* HTMLPage (método *dependencies*) e utilizando os métodos *include_js* e *include_css* para efetuar a adição das bibliotecas javascript necessárias na instância de HTMLPage.

4.2.8.1.4. Comunicação entre *widgets* pai e filho

Certos *widgets* concretos precisam ter acesso as instâncias dos *widgets* superiores (pais) para consultar valores ou até mesmo alterar valores nesses *widgets*. Isso ocorre mais comumente em famílias de *widgets* criadas para serem utilizadas em conjunto.

Para que essa comunicação entre *widgets* pai e filho seja possível, todo *widget* tem acesso a uma variável de instância chamada *@parent*, que retorna a instância do *widget* pai, podendo assim manipular os dados do *widget* pai através de seus métodos.

Um exemplo disso é o *widget* DHTMLXSchedulerEntry que, ao invés de renderizar uma interface, insere um conjunto de dados *(@entries*) para o *widget* pai (Quadro 29) que é uma instância do *widget* DHTMLXSchedulerComposition.

```
def render
  @parent.add_entry(@entries) if @parent.respond_to?(:add_entry)
end
```

Quadro 29 – Widget DHTMLXSchedulerEntry utilizando a variável @parent

4.2.8.1.5. Arquivo de manifesto (assinatura do Widget Concreto)

Para o projetista compreender qual a descrição de um *widget* concreto, sua versão, quais são os parâmetros disponíveis, com que *widgets* abstratos poderá ser mapeado e exemplos de regras de mapeamento, cada *widget* possui um arquivo de descrição de suas características chamado arquivo de manifesto.

As principais propriedades descritas nesse arquivo são:

- **name** – nome do *widget* concreto;
- **version** – versão do *widget*;
- **description** – Texto informado o que o *widget* faz;
- **compatible_abstracts** – Informa que elementos abstratos esse *widget* mapeia;
- **dependencies** – Lista de dependências com outros *widgets*;
- **parameters** – lista de parâmetros aceitos pelo *widget*;
- **examples** – exemplos de mapeamento concreto com o *widget*.

Cada *widget* deverá possuir em sua pasta um arquivo chamado MANIFEST.rb, e em seu conteúdo a descrição e feita por um Hash Ruby.

O Quadro 30 apresenta o arquivo de manifesto do *widget* HTMLAnchor.

```
{
  name: "HTMLAnchor",
  version:  "0.0.1",
  description: "HTML Anchor",
  compatible_abstracts: [ "SimpleActivator" ],
  dependencies: [ ],
  parameters: [
    {name: "content", mandatory: true, data_type: "string" },
    {name: "url", mandatory: true, data_type: "string"},
    {name: "css_class", mandatory: false, data_type: "string"},
    {name: "id", mandatory: false, data_type: "string"},
  ],
  examples: [
    %q{

    maps_to abstract: "link1", concrete_widget: "HTMLAnchor",

    params: {  content: 'My link anchor', url: '/mypage.html' }

    maps_to abstract: "link2", concrete_widget: "HTMLAnchor",

    params: {  content: variable.label, url: variable.url } do

      equal variable.label, "Home"

    end}
  ]
}
```

Quadro 30 – Arquivo de manifesto do *widget* concreto HTMLAnchor

4.2.8.1.6. Estrutura de arquivos dos Widgets Concretos

Um *widget* concreto é armazenado no diretório *concrete-widget* do interpretador de interface. Um padrão de nomenclatura deve ser seguido para criar um *widget*:

- O nome do diretório deve ser o nome do *widget* criado em padrão *camel-case*[54];
- O arquivo da classe Ruby deve ser o mesmo do nome do diretório;
- O nome da classe nesse arquivo também deve ser o mesmo do diretório;
- Caso utilize template Tilt, esse deve ter o nome template.[ext], sendo a extensão correspondente ao tipo de template desejado. Por exemplo, para templates Erubis **template.erb**, para templates Markaby **template.mab**;
- O arquivo de manifesto deve se chamar MANIFEST.rb

Por exemplo, o *widget* HTMLAnchor possui seu diretório estruturado conforme o Quadro 31.

```
/HTMLAnchor
  HTMLAnchor.rb
  template.erb
  MANIFEST.rb
```

Quadro 31 – Diretório do *widget* concreto HTMLAnchor

Uma extensão também é estruturada da mesma maneira que um *widget* concreto com a diferença que fica armazenada na pasta *extensions* do interpretador.

[54] http://pt.wikipedia.org/wiki/CamelCase

4.2.8.2. Máquina de interpretação e renderização

A seção 4.1.1.5 apresentou os princípios da arquitetura de construção de uma máquina de interpretação e renderização de interfaces utilizando *widgets* concretos. Esse tópico apresenta um interpretador escrito em linguagem Ruby capaz de renderizar interfaces através de um esquema de interface.

O esquema de interface compatível com o interpretador implementando é uma estrutura em árvore representada por um Hash Ruby. O avaliador concreto incluído na arquitetura apresentada na seção 4.2.2 faz uso desse esquema e tem como saída exatamente uma estrutura em hash compatível com o interpretador implementado.

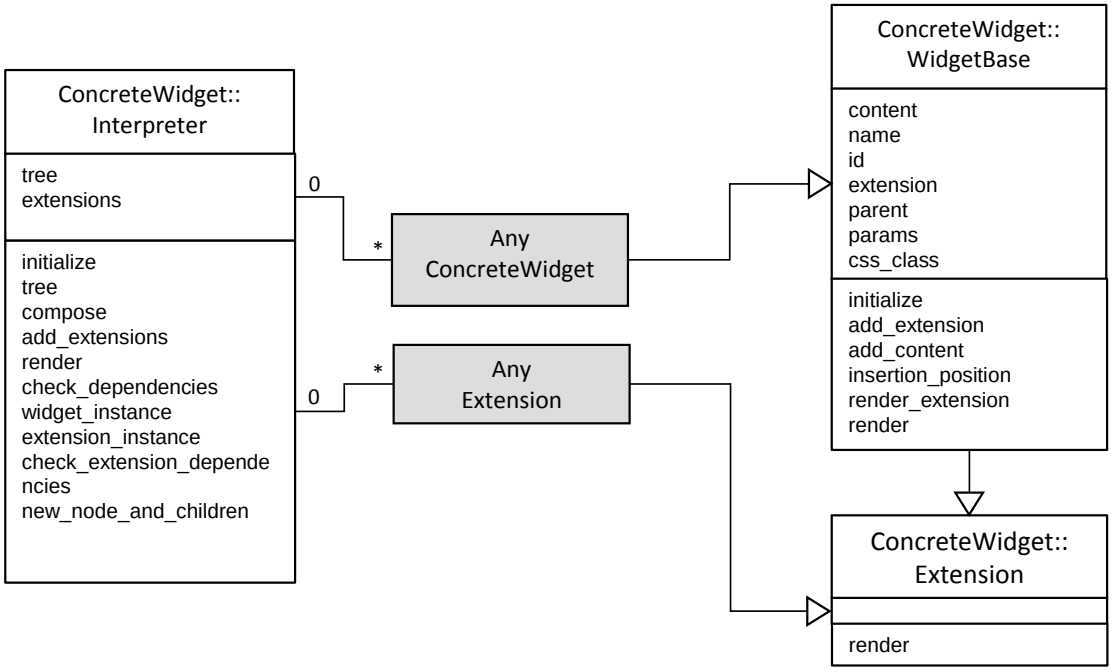

Figura 22 – Diagrama de classes da máquina de interpretação

A implementação do interpretador é feita através da classe ConcreteWidget::Interpreter. O método construtor dessa classe, ao receber o hash com o esquema da interface, é responsável por invocar o método compose que, por sua vez, percorre o hash do esquema e para cada elemento instancia um nó da estrutura da árvore da interface.

Para implementar a estrutura em árvore, foi utilizada uma biblioteca chamada RubyTree[55]. A RubyTree fornece uma API genérica para manipular

[55] http://rubytree.rubyforge.org/rdoc/

estruturas em árvore, tanto para acesso direto aos nós da árvore, através de chaves de acesso, quanto para alterar e percorrer seus elementos.

Para cada nó da árvore, será criada uma instância do *widget* concreto (método widget_instance) correspondente, recebendo como argumento os parâmetros fornecidos no hash.

Com a árvore completamente instanciada, será possível adicionar as extensões concretas (método add_extensions), que são implementadas como um array de hashs.

Para renderizar a interface, a árvore será percorrida por recursão (método render) e ao encontrar os *nós folhas* da árvore, cada *widget* terá seu método render invocado e esses métodos retornarão o resultado da sua renderização. Os resultados fragmentados de renderização são concatenados aos do atributo content de cada nó superior da estrutura (nó pai) até alcançar a raiz. Durante essa recursão cada *widget* também resolve as suas dependências já renderizando o resultado esperado.

Para gerar um hash compatível com o interpretador, cada elemento ou nó da estrutura deve possuir as seguintes propriedades, implementadas como chaves do hash:

- **name** – identificação do nó. Deve ser único em toda a estrutura;
- **node_content** – hash que possui outras duas sub-propriedades:
  - **concrete_widget** – nome do *widget* concreto que vai ser renderizado para o elemento;
  - **params** – parâmetros de renderização enviados para a instância do *widget* concreto;
- **children** – array com a estrutura dos elementos filhos do nó corrente.

O Quadro 32 apresenta um pequeno exemplo de esquema de interface utilizando as propriedades citadas anteriormente.

```
{ :name => 'main_page', :node_content => {
    :concrete_widget => "Page",
    :params => {:title => "Demo page", :include_css=> ["screen.css"] }
   },
  :children => [
    { :name => 'page_content',
      :node_content => {:concrete_widget => "HTMLComposition"},
      :children => [
        { :name => "first_heading",
          :node_content => { :concrete_widget => "HTMLHeading",
```

```
                :params => {
                :content => "First heading", :css_class => "heading2"}
                }
          }
        ]
    }
}
```

Quadro 32 – Exemplo de esquema de interface para interpretador concreto

Para declarar a lista de extensões, um array deve ser definido onde cada elemento é um hash contendo as seguintes propriedades:

- **name** – identificação da extensão;
- **extension** – nome da extensão ou *widget* concreto desejado;
- **nodes** – array com a lista de elementos que serão extendidos, sendo o mesmo utilizado no esquema da interface;
- **params** – parâmetros repassados para a extensão.

O Quadro 33 demonstra um exemplo de array de extensões, onde o primeiro utiliza o *widget* concreto HTMLLineBreak e o segundo a extensão JQueryFormAjax.

```
[
{:name => 'ext1', :extension => 'HTMLLineBreak',
  :nodes =>   ['node1', 'node2']},
 {:name => 'ext2', :extension => 'JQueryFormAjax',
  :nodes => ['form_search'], :params => {:target => "tweets"}}
]
```

Quadro 33 – Exemplo de array de extensões de interface

A complexidade da interface depende da escolha dos *widgets* concretos que possuam as capacidades desejadas pelo projetista e a composição enviada para o interpretador.

Um exemplo completo de esquema de interface compatível com a máquina de renderização e capaz de fazer buscas de expressões no Twitter[56] está disponível no Apêndice II.

[56] https://twitter.com/

4.3.
Ambiente de autoria

O Synth foi o ambiente de modelagem de aplicações utilizado no desenvolvimento desse trabalho. O Synth possui um ambiente de autoria de aplicações que necessitou ser atualizado para que suportasse o novo módulo de interface.

> O ambiente de autoria do Synth dispõe de uma interface gráfica de formulários que é acessada por meio de um navegador de internet e permite a execução, pelo projetista, das operações de visualização, criação, remoção e edição das primitivas dos modelos de uma aplicação construída segundo o método SHDM. Através desta interface é possível executar a aplicação enquanto ocorre a sua construção e, dessa forma, validá-la a cada passo do seu desenvolvimento.
>
> [Bomfim, 2011]

O ambiente do Synth oferece suporte para modelagem de todas as etapas do método SHDM que são: a modelagem de domínio, o projeto navegacional, o projeto comportamental e finalmente o projeto de interface.

Nesse trabalho, o ambiente de modelagem de interfaces existente originalmente no Synth foi removido, dando lugar para um novo módulo que dá suporte as etapas de modelagem de interface segundo o método proposto nesse trabalho. Esse módulo possui uma interface de autoria que divide a modelagem em três etapas: as **regras de seleção de interface**, a **descrição abstrata,** as **regras de composição** e o **mapeamento concreto**.

Todos os demais módulos do ambiente de autoria do Synth foram mantidos, não sendo o foco desse trabalho apresentá-los com detalhes. Para um maior aprofundamento sugere-se a leitura da seção 4.4 do trabalho de Bomfim [2011] que explica cada módulo disponível no ambiente de autoria do Synth.

4.3.1.
Tela inicial

Ao acessar o Synth por seu endereço padrão [http://localhost:3000] teremos acesso a tela de acionamento das aplicações existentes, podendo alternar entre as existentes ou criar uma nova, conforme a Figura 23.

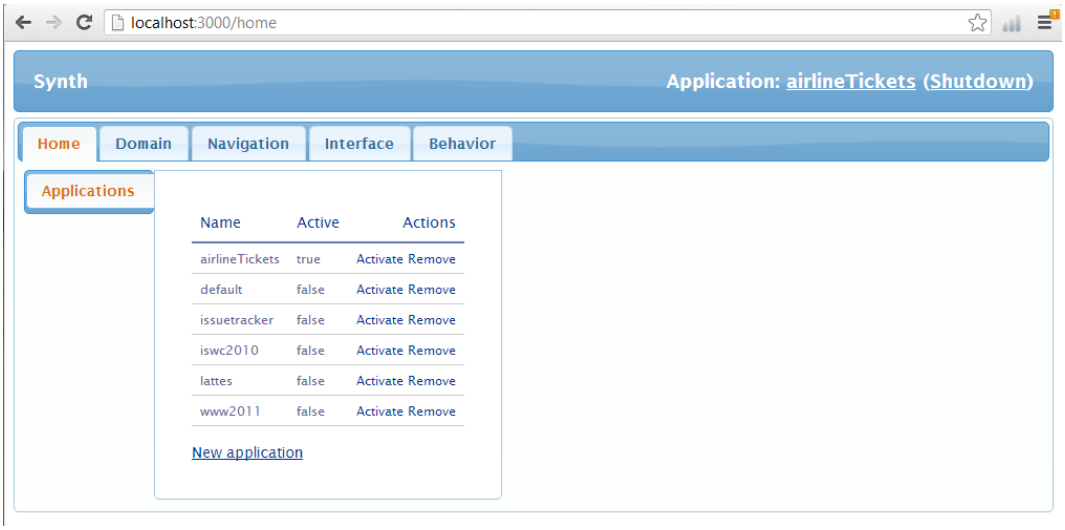

Figura 23 - Página inicial do Synth

Sempre estarão disponíveis, na parte superior, abas de acessos aos demais módulos do Synth que são: Domain, Navigation, Behavior e Interface.

4.3.2.
Modelagem de domínio

Através do endereço [http://localhost:3000/domain] ou pelo menu superior, teremos acesso à modelagem de domínio, tal como a Figura 24 ilustra. Essa tela dá acesso às seguintes áreas: “Ontologies”, “Namespaces”, “Datasets”, “Resources”, “Classes” e “Properties”.

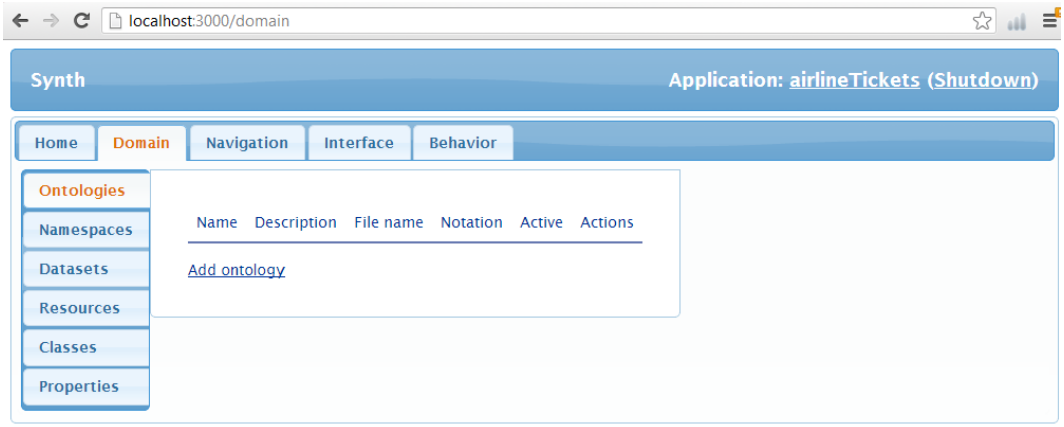

Figura 24 - Tela de modelagem de domínio

Através das áreas internas é possível dar carga de ontologias, definir namespaces para os recursos, visualizar e alterar classes, propriedades e recursos RDF.

4.3.3. Projeto navegacional

Na tela de projeto navegacional (Figura 25) é possível criar contextos a partir das classes de domínio, índices de navegação, *Landmarks* e definir novas propriedades para classes em contexto.

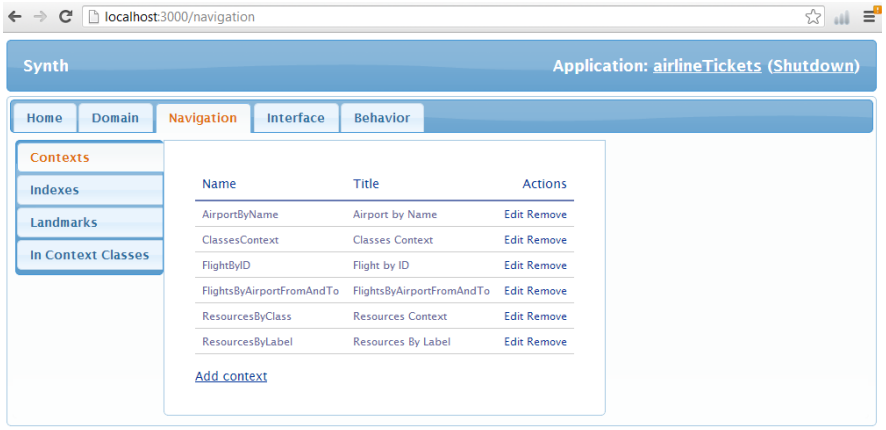

Figura 25 - Tela de modelagem navegacional

4.3.4. Projeto comportamental

Na interface do projeto comportamental é possível criar as operações internas e externas. A Figura 26 ilustra uma lista de operações de uma aplicação.

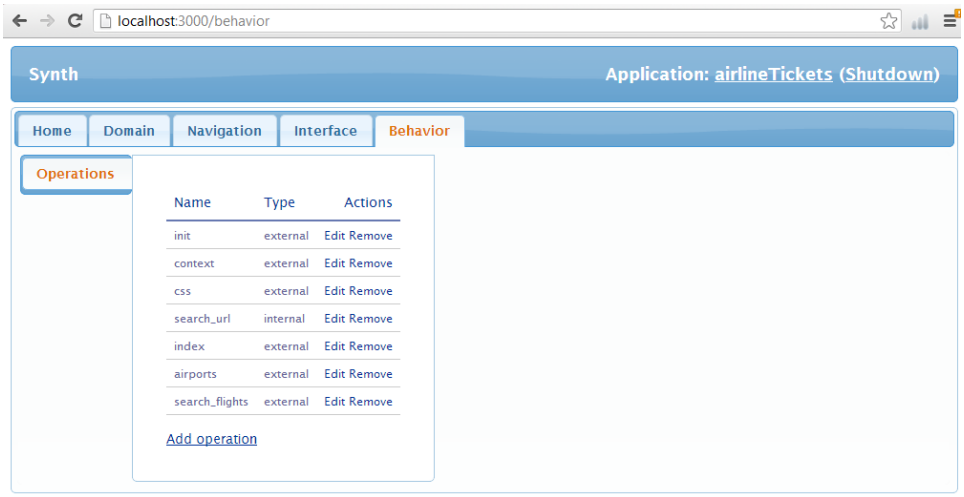

Figura 26 - Tela de modelagem comportamental

Uma operação externa é responsável por receber os parâmetros de navegação, gerar as instâncias dos objetos do modelo navegacional, preparar os fatos e os dados de interface e invocar a renderização.

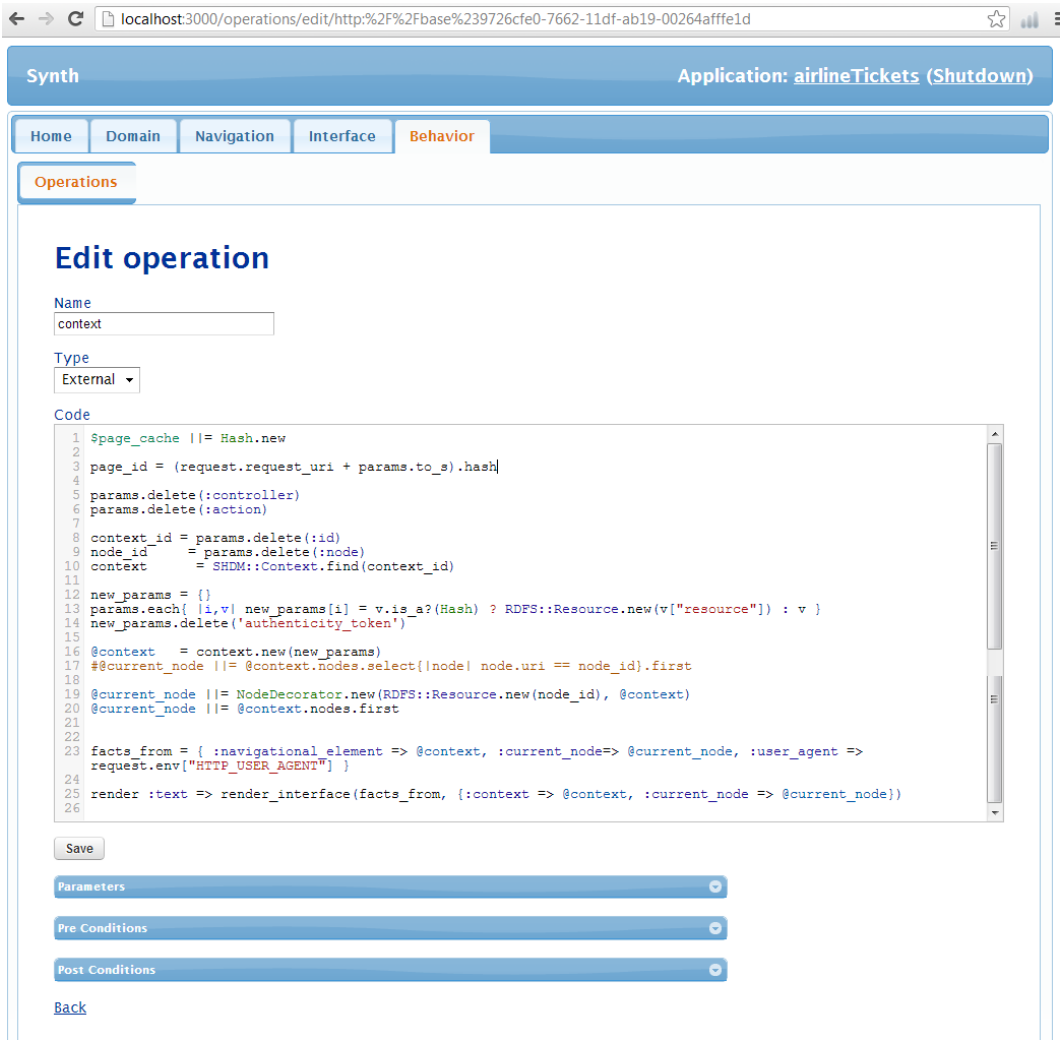

Figura 27 - Edição de operação externa

A operação externa é acessada pelo endereço no formato [http://nome_do_servidor/execute/nome_da_operação] e a Figura 27 ilustra um exemplo de operação externa para um nó em contexto.

4.3.5. Projeto de interfaces

A tela que dá acesso ao projeto de interfaces exibe inicialmente uma lista de interfaces existentes numa aplicação, além do acesso para criação de uma nova interface. Para cada interface da lista é exibido o nome, tipo, peso e as ações de editar e remover a interface, conforme ilustrado pela Figura 28.

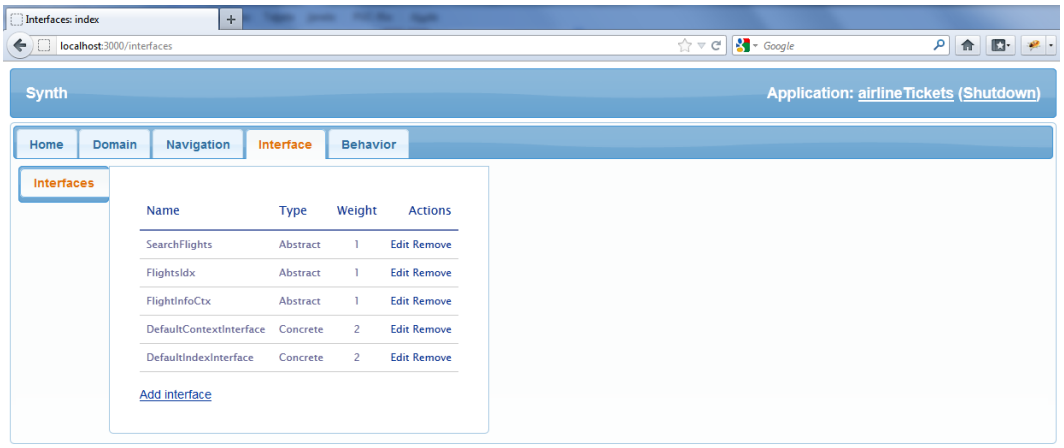

Figura 28 - Lista de interfaces

Apesar da arquitetura de interfaces anterior ter sido removida, uma funcionalidade foi reimplementada para manter a compatibilidade com interfaces descritas diretamente em código Ruby ou com templates ERB. Isso foi feito por existir um legado de interfaces descritas dessa maneira e assim elas continuarem funcionando no Synth, porém acrescidas das regras de seleção de interfaces. A propriedade *type* foi incluída para diferenciar as interfaces descritas diretamente em Ruby (*concrete*) das que utilizam a arquitetura proposta nesse trabalho (*abstract*).

4.3.5.1. Tela de edição

Quando uma interface nova é criada, dois atributos precisam ser fornecidos como dados gerais: nome (*name*) e o título (*title*). A tela de criação exibe um formulário com esses campos, conforme a Figura 29.

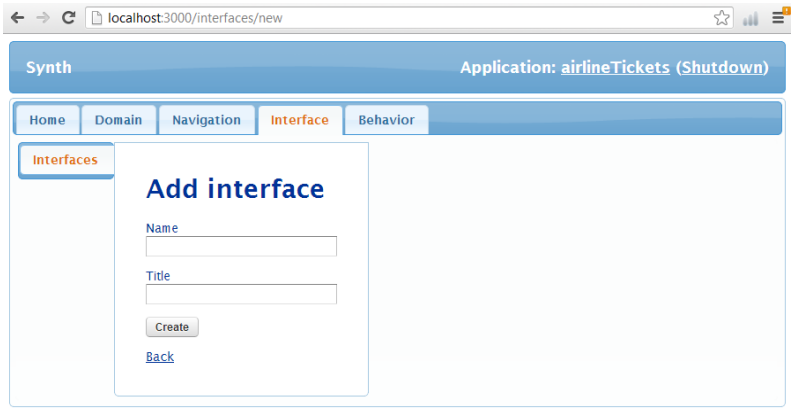

Figura 29 - Criando uma nova interface

Após a confirmação de criação de uma interface, será exibida a tela de edição de interface, que além dos dados gerais (*General Data*) exibe opções de acesso para três etapas da modelagem: as regras de seleção de interface (*Interface Selection Rules*), a descrição abstrata e as regras de composição (*Abstract Description and Rules*) e o mapeamento concreto (Concrete Mapping). A Figura 30 ilustra as opções de acesso para cada etapa.

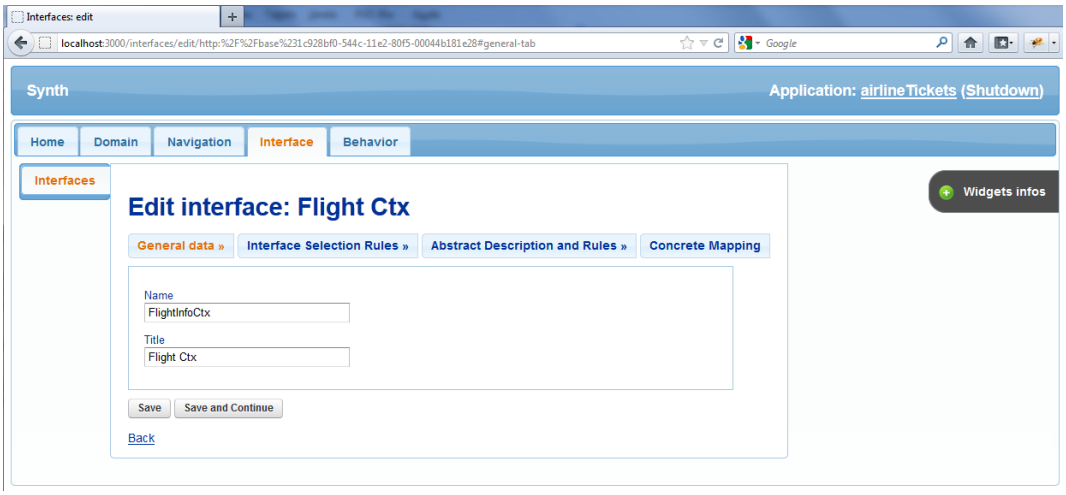

Figura 30 - Tela de edição de interface

Para cada etapa sempre estará disponível o botão *save*, que salva os dados da interface e retorna para lista de interfaces; o botão *save and continue*, que salva os dados e continua na edição da interface e a ancora *back*, que retorna para a lista de interfaces descartando qualquer alteração.

4.3.5.2. Regras de seleção de interface

Ao acionar a opção *Interface Selection Rules* serão exibidos os campos de:

- ***Interface weight*** – onde um peso para a interface deverá ser escolhido. Quanto menor o peso, maior a prioridade;
- ***Interface selection rule*** – recebe de maneira declarativa as regras de seleção da interface conforme a DSL descrita na seção 4.2.5;
- ***Description type*** – para escolha de qual o tipo da descrição da interface entre as opções *abstract* e *concrete*.

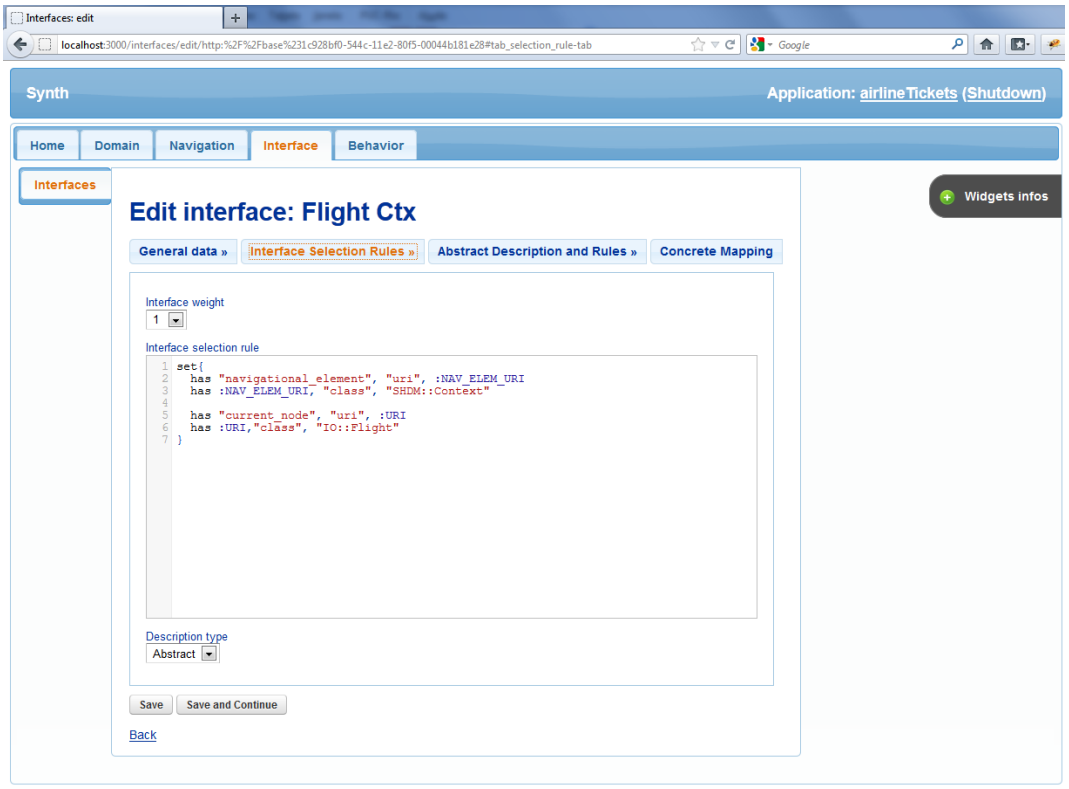

Figura 31 - Tela para regras de seleção de interface

Para facilitar a escrita das regras de seleção foi inserido um editor web[57] com realce de sintaxe Ruby (*syntax highlighting*), numeração de linhas e controle de tabulação conforme pode ser visto na Figura 31.

Se nenhuma regra for fornecida a interface não será selecionada.

O mecanismo de regras de seleção também será aplicado para interfaces do tipo concreto.

4.3.5.3. Descrição abstrata e as regras de composição

Ao selecionar a opção *Abstract Description and Rules*, serão exibidos dois campos que recebem de forma declarativa:

- **Abstract description** – o hash da descrição abstrata da interface conforme a sintaxe de declaração da seção 4.2.6.1;
- **Abstract composition rules** – as regras de seleção de elementos abstratos conforme a DSL de declaração da seção 4.2.6.2.

[57] O editor de código utilizado foi o Codemirror - http://codemirror.net

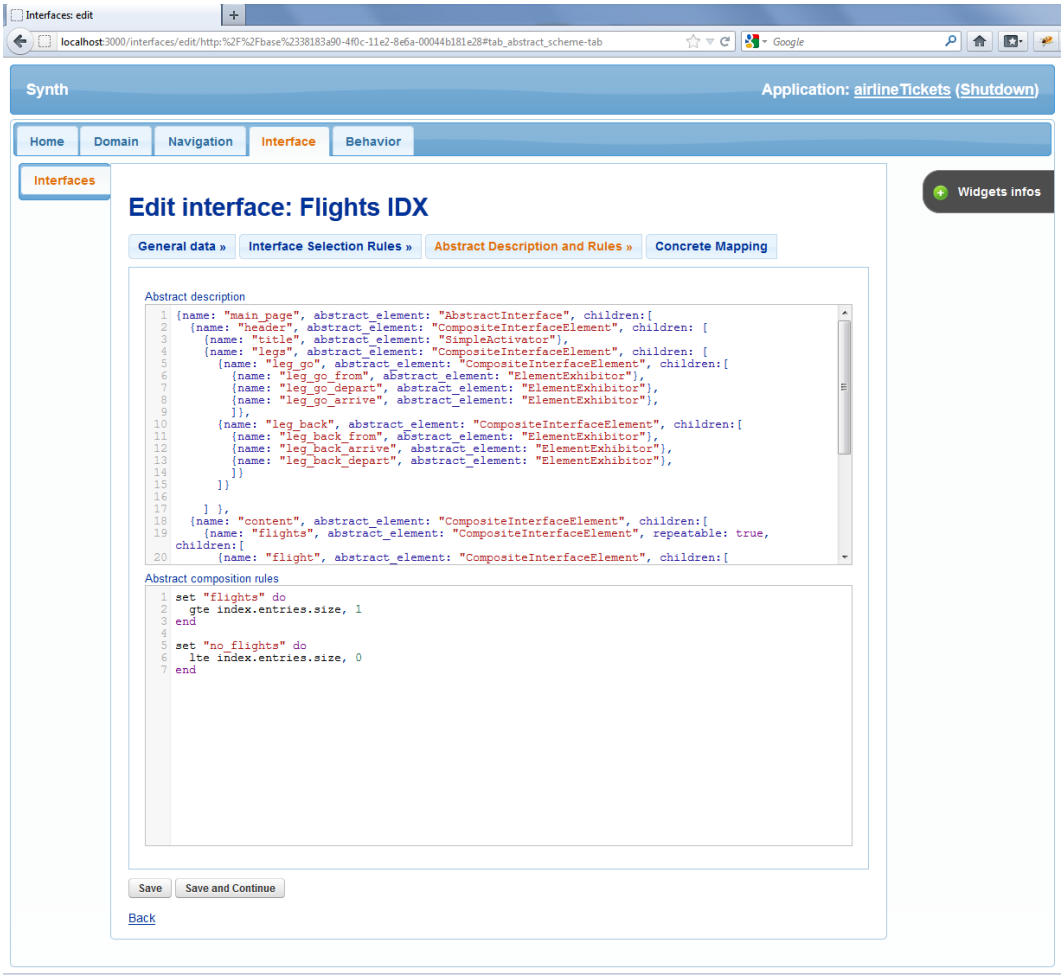

Figura 32 - Tela de composição abstrata e regras de seleção

Ambos os campos dessa etapa da modelagem utilizam o editor com realce de sintaxe, conforme Figura 32.

4.3.5.4. Mapeamento concreto

A opção *Concrete mapping* dá acesso à última etapa de modelagem de uma interface. Estão disponíveis dois campos que são descritos conforme a DSL da seção 4.2.7:

- **Concrete mapping rules** – para declarar as regras de mapeamento concreto;
- **Concrete extensions** – para declarar as extensões concretas de interface.

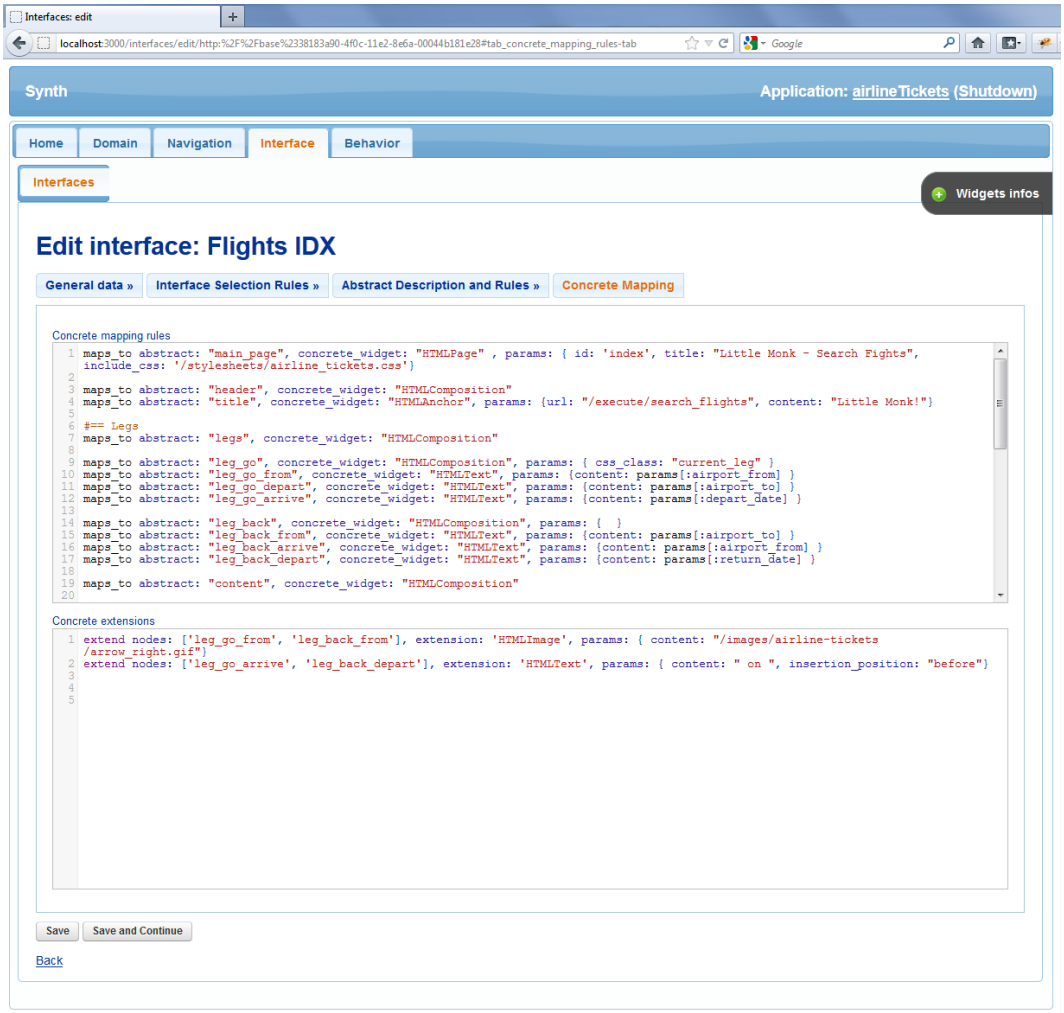

Figura 33 - Declaração de regras de mapeamento concreto

Conforme as regras de mapeamento concreto forem sendo declaradas, o resultado da renderização poderá ser visualizado executando a operação correspondente.

4.3.5.5. Informações dos *widgets*

Para auxiliar o projetista na escolha e uso dos *widgets* concretos, uma janela flutuante está disponível através da opção *Widget infos*. Essa janela possui uma lista com todos os *widgets* concretos e extensões disponíveis. Ao selecionar um dos *widgets* da lista serão exibidas as informações de seu arquivo de manifesto, conforme a Figura 34.

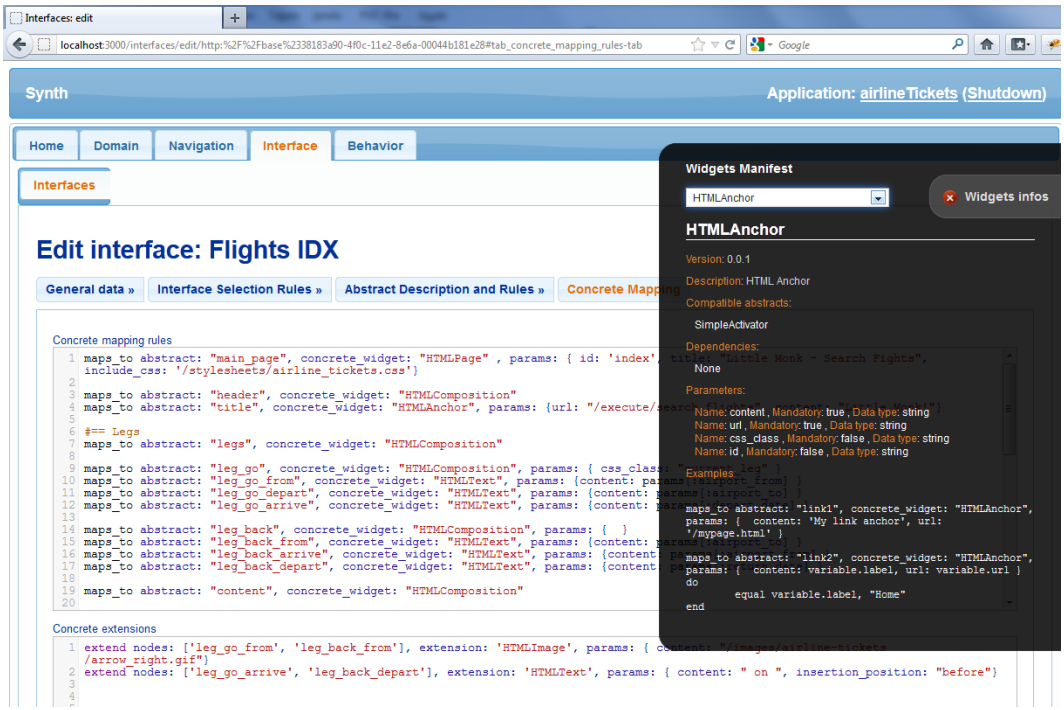

Figura 34 - Janela de informações de Widgets

Com essas informações o projetista poderia saber quais são os parâmetros disponíveis de cada *widget*, que elemento abstrato é compatível e consultar alguns exemplos de regras de mapeamento concreto.

5
Exemplos

Esse capítulo apresenta um exemplo de aplicação modelada no ambiente Synth incluindo os modelos de domínio, navegação e comportamental do SHDM, além dos modelos de interfaces desenvolvidos nesse trabalho[58].

O exemplo é uma aplicação simulando buscas de passagens em uma empresa aérea fictícia. Todos os dados dos voos apresentados foram gerados e importados para o Synth a partir de arquivos RDF[59]. Chamaremos a aplicação de exemplo criada de "Airline Tickets".

Para essa aplicação foram projetadas interfaces para atender três situações simples de uso:

- **Seleção de aeroportos** - interface para o usuário informar os aeroportos de partida e destino e a data de ida e volta do voo;
- **Lista de voos** - resultado com a lista de voos localizados para o destino escolhido;
- **Detalhes do voo** - exibe mais detalhes do voo.

O objetivo da aplicação *Airline Tickets* é exemplificar, utilizando um modelo navegacional e de dados simplificado, a construção de interfaces que utilizam componentes ricos software como resultado da modelagem e execução através da arquitetura apresentada no capítulo anterior e de *widgets* concretos que possuem a capacidade de expressar elementos de interação com a complexidade exigida pelo projetista.

[5858] O exemplo modelado está acessível juntamente com o ambiente Synth em https://github.com/vagnernascimento/synth/tree/interface_rules

[59] O arquivo RDF com os dados dos voos simulados foi adquirido do exemplo de aula de García [2010], sendo esses acrescidos com propriedades para identificar os aeroportos pelo código IATA.

5.1.
Modelagem de domínio

O modelo de domínio da aplicação *Airline Tickets* possui somente duas classes: *Airport* e *Flight*. As propriedades disponíveis de cada classe são as listadas no diagrama UML da Figura 35.

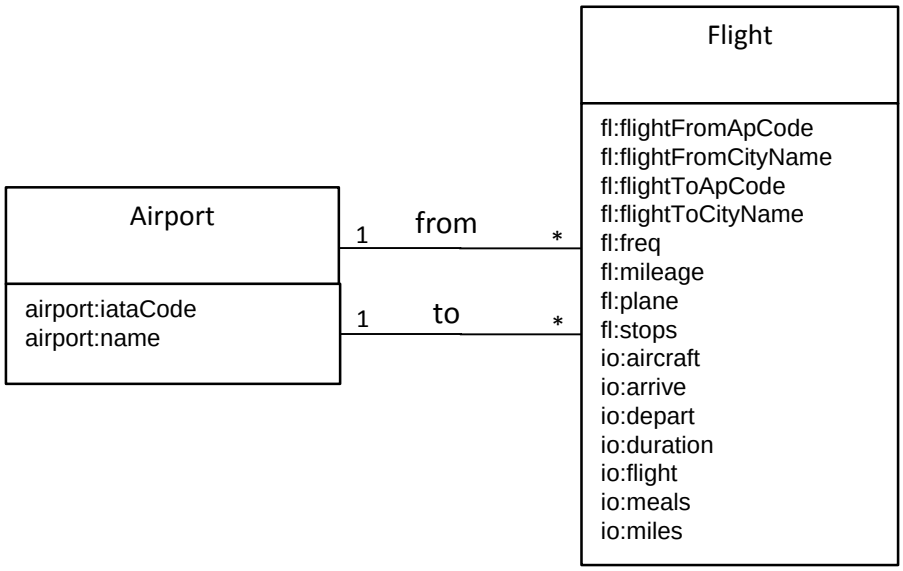

Figura 35 – Diagrama de classes de domínio da aplicação Airline Tickets

A classe *Airport* possui propriedades para o código IATA[60] e o nome do aeroporto, conforme exemplo da Figura 36.

Properties	
uri	http://www.daml.ri.cmu.edu/ont/AirportCodes.daml#HKN
airport:iataCode +	"HKN"
airport:name +	"Hoskins, Papua New Guinea"
rdf:type +	airport:Airport rdfs:Resource

Figura 36 – Exemplo de dados de um aeroporto

A classe *Flight* possui propriedades para informações entre um voo de um aeroporto de origem e destino, como pode ser visto no exemplo da Figura 37.

[60] http://www.iata.org

Properties	
uri	fl:SX0106
fl:flightFromApCode +	http://www.daml.ri.cmu.edu/ont/AirportCodes.daml#ABZ
fl:flightFromCityName +	"Aberdeen GB"
fl:flightToApCode +	http://www.daml.ri.cmu.edu/ont/AirportCodes.daml#LHR
fl:flightToCityName +	"London-Heathrow GB"
fl:freq +	"x67"
fl:stops +	"0"
io:aircraft +	"320"
io:arrive +	"11:10a"
io:depart +	"9:40a"
io:duration +	"1:30"
io:flight +	"SX0106"
io:meals +	"M"
io:miles +	"402"
rdf:type +	io:Flight rdfs:Resource

Figura 37 – Exemplo de valores para as propriedades de um voo da classe *Flight*

5.2. Projeto navegacional

O projeto navegacional da aplicação *Airline Tickets* possui um contexto para cada classe citada na modelagem de domínio, conforme ilustrado na Figura 38.

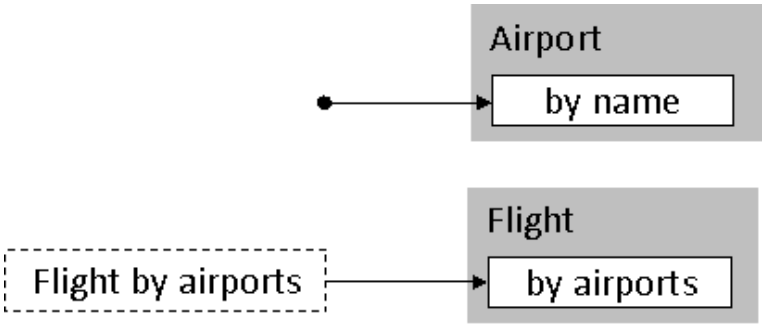

Figura 38 – Contextos da aplicação AirlineTickets

Ambos os contextos da aplicação são parametrizados. O contexto *by name* da classe *Airport* necessita que o parâmetro *name* seja fornecido para retornar o resultado da consulta. O contexto *by airports* da classe *Flight* espera os parâmetros *airport_from* e *airport_to* para executar a consulta.

```
#= AIRPORT BY NAME
selects {
  a AIRPORT::Airport
  airport::name like name
limit 30
}

#== FLIGHT BY AIRPORTS
selects {
  a IO::Flight
  fl::flightFromApCode airport_from
  fl::flightToApCode airport_to
}
#== FLIGHT BY AIRPORTS
selects {
  a IO::Flight
  fl::flightFromApCode airport_from
  fl::flightToApCode airport_to
}
```

Quadro 34 – Consultas para os contextos da aplicação Airline Tickets

O Quadro 34 demonstra as consultas para os contextos *by name*, utilizando a sintaxe em SynthQL [Bomfim, 2011].

Apenas um índice de navegação foi definido na aplicação, chamado de *FlightsByAirportsIdx*, que é o índice de elementos do contexto *by airports* da classe *Flight*. Esse índice retorna a lista de resultados dos voos por aeroportos de partida e chegada acrescidos de atributos computados e uma âncora para o contexto. O Quadro 35 apresenta uma representação do índice e seus atributos em notação RDF.

```
:FlightsByAirportsIdx a shdm:ContextIndex ;
  shdm:index_name "FlightsByAirportsIdx";
  shdm:index_title "Flights by Airports Idx";
  shdm:context_index_context :FlightsByAirports;
  shdm:context_anchor_attributes [
    a shdm:ContextAnchorNavigationAttribute;
      shdm:navigation_attribute_name "context_anchor";
      shdm:navigation_attribute_index_position "1";
      shdm:context_anchor_label_expression "self.rdfs::label";
      shdm:context_anchor_target_context :FlightsByAirports;
      shdm:context_anchor_target_node_expression "self";
      shdm:index_navigation_attribute_index_parameters [
        a shdm:NavigationAttributeParameter;
          shdm:navigation_attribute_parameter_name "airport_from";
          shdm:navigation_attribute_parameter_value_expression
"parameters[:airport_from]".
        a shdm:NavigationAttributeParameter;
          shdm:navigation_attribute_parameter_name "airport_to";
          shdm:navigation_attribute_parameter_value_expression
"parameters[:airport_to]".
        a shdm:NavigationAttributeParameter;
          shdm:navigation_attribute_parameter_name "depart_date";
          shdm:navigation_attribute_parameter_value_expression
"parameters[:depart date]".
        a shdm:NavigationAttributeParameter;
          shdm:navigation_attribute_parameter_name "return_date";
```

```
          shdm:navigation_attribute_parameter_value_expression
"parameters[:return_date]".
      ].
  ];
  shdm:computed_attributes [
    a shdm:ComputedNavigationAttribute;
    shdm:navigation_attribute_name "start_date";
    shdm:navigation_attribute_index_position 1;

    shdm:computed_value_expression '
      t = self.io::depart.to_s.match(/^([\d]{1,2}:[\d]{2}[ap]).*/)
      DateTime.parse( "#{parameters[:depart date]} #{t[1]}m").strftime("%Y-%m-%d
%H:%M")'.
    a shdm:ComputedNavigationAttribute;
    shdm:navigation_attribute_name "end_date";

    shdm:navigation_attribute_index_position 2;

    shdm:computed_value_expression '
      ta = self.io::arrive.to_s.match(/^([\d]{1,2}:[\d]{2}[ap])(.*)/)
      arrive = DateTime.parse( "#{parameters[:depart_date]} #{ta[1]}m")
      arrive = eval( "arrive #{ta[2]}") unless ta[2].nil?
      arrive.strftime("%Y-%m-%d %H:%M")'.

    a shdm:ComputedNavigationAttribute;
    shdm:navigation_attribute_name "price";
    shdm:navigation attribute index position 3;

    shdm:computed_value_expression '
      price = self.io::miles.to_s.to_i * 0.1 + (500 / (self.duration.label * 1.5)
)
      "#{"%.2f" % (price)}"'.

    a shdm:ComputedNavigationAttribute;
    shdm:navigation_attribute_name "text";
    shdm:navigation_attribute_index_position 4;

    shdm:computed_value_expression '
      ta = self.io::arrive.to_s.match(/^([\d]{1,2}:[\d]{2}[ap])(.*)/)

      "#{self.io::depart.to_s}m - <b>#{self.io::flight.to_s} </b>
#{ta[1]}m#{ta[2]}"'.
  ].
```

Quadro 35 - Índice FlightsByAirportsIdx em RDF notação turtle

5.3.
Projeto comportamental

No projeto comportamental da aplicação foram criadas três operações externas: *airports*, *search_flights* e *context*.

A operação *airports* retorna em formato *Json*, uma lista de aeroportos filtrada pelo parâmetro *search.* Além disso, caso algum valor seja recebido através do parâmetro *airport_from*, esse mesmo será procurado e removido da lista de aeroportos retornada. O código Ruby da operação pode ser visto no Quadro 36.

```
context     =
SHDM::Context.find_by.context_name("AirportByName").execute.first
new_params = {name: params[:search]}
context_inst = context.new(new_params)
result = []
```

```
context_inst.resources.delete_if{ | item |
  item.iataCode.to_s == params[:airport_from] }
context_inst.resources.each{ | item |
    result << { label: "#{item.name}, #{item.iataCode}",
                value: item.iataCode.to_s }
}
render :json => result
```

Quadro 36 – Operação *airports* para retornar lista de aeroportos em Json

A operação *search_flights* é responsável por invocar a renderização de interfaces para duas situações: o formulário para busca de voos e o índice de resultados com os voos encontrados.

```
facts = {
  :user_agent => request.env["HTTP_USER_AGENT"],
  :params => params,
}
interface_data = {:params => params}

if params[:airport_from] and params[:airport_to]
  index =
SHDM::Index.find_by.index_name("FlightsByAirportsIdx").execute.fir
st
  @index = index.new(params)
  facts[:navigational_element] = @index
  best_price = @index.entries.min_by{ | x | x.price.label }
  interface_data = {:params => params, :index => @index,
                    :best price => best price }
end

render :text => render_interface(facts, interface_data)
```

Quadro 37 – Operação *search_fights* invoca a renderização de interfaces

Conforme ilustrado no Quadro 37, um Hash contendo o *user_agent* do usuário e os parâmetros de navegação serão utilizados como entrada para geração de fatos para as regras enviadas para o interpretador de interfaces, tal como visto na seção 4.2.3.2. Os parâmetros de navegação também serão utilizados como dados da interface. Caso existam valores para os parâmetros *airport_from* e *airport_to*, o índice *FlightsByAirportsIdx* será instanciado, sendo esse incluído tanto como dados das regras (*navigational_element*) e dados de mapeamento (*index*).

A última operação utilizada na aplicação é uma operação padrão do Synth para instanciar e renderizar um recurso em um contexto de navegação, que é chamada de *context*.

```
params.delete(:controller)
params.delete(:action)

context_id = params.delete(:id)
```

```
node_id     = params.delete(:node)
context      = SHDM::Context.find(context_id)

new_params = {}
params.each{ |i,v| new_params[i] = v.is_a?(Hash) ?
RDFS::Resource.new(v["resource"]) : v }
new_params.delete('authenticity_token')

@context    = context.new(new_params)
@current_node = @context.nodes.select{|node| node.uri == node_id}.first
@current_node ||= NodeDecorator.new(RDFS::Resource.new(node_id),
@context)

params[:node] = node_id
facts_from = {
  :navigational_element => @context,
  :current_node=> @current_node,
  :user_agent => request.env["HTTP_USER_AGENT"],
  :params => params
}
interface_data = {
  context: @context, current_node: @current_node
}
render :text => render_interface(facts_from, interface_data)
```

Quadro 38 – Operação *context* é padrão nas aplicações do Synth

O código Ruby da operação *context*, descrita no Quadro 38, assim como na operação *search_flights* possui em sua descrição a definição dos dados de regras (Hash *facts*), dos dados de mapeamento (Hash *interface_data*) e o método para invocar a interface renderizada.

5.4. Projeto de interfaces

Veremos a seguir o projeto de três interfaces da aplicação Airline Tickets: Uma para busca e seleção dos aeroportos, outra para exibição do índice de voos localizados e a última para exibição de detalhes de um voo. Para cada uma delas veremos suas regras de seleção, composição abstrata (e suas regras), regras de mapeamento concreto e a lista de extensões concretas. Em seguida, veremos alterações nos modelos das interfaces apresentadas para uma possível versão para dispositivos móveis (*mobile*).

5.4.1. Search Flights

A interface chamada de *search flights* foi projetada para exibir um formulário para entrada dos dados para busca de voos. A Figura 39 ilustra o

resultado da renderização gerado pela máquina de interpretação e em seguida será apresentada cada etapa da modelagem dessa interface.

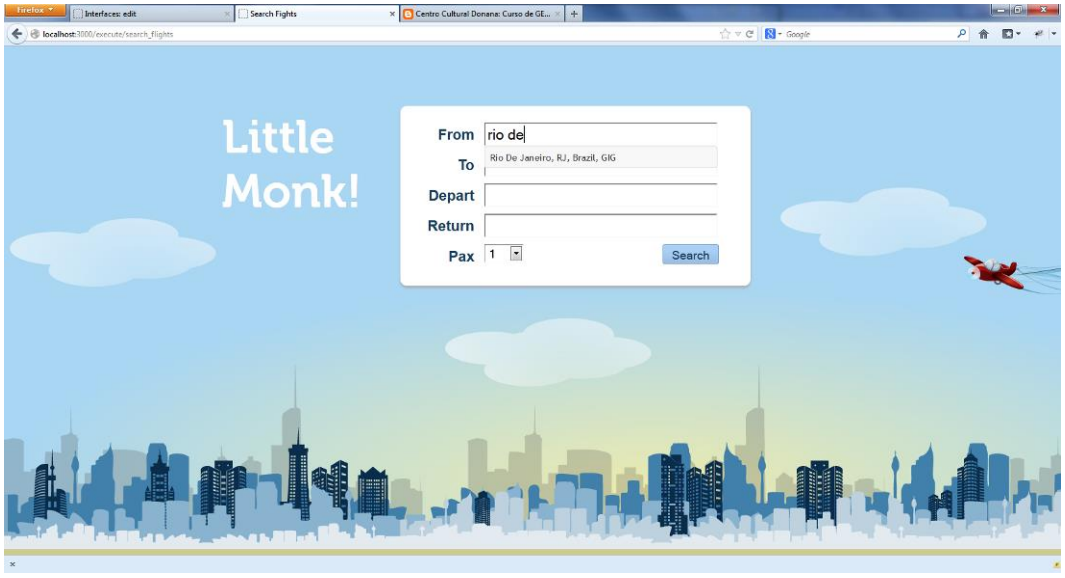

Figura 39- Interface search flights renderizada

Seleção da interface

Quando a operação externa *search_flights* invoca uma interface enviando um conjunto de fatos, o conversor de fatos gera o conjunto de triplas indicado no Quadro 39 para alimentar a base de conhecimento da máquina de regras.

```
<WME "user_agent" "browser" "Chrome">
<WME "user_agent" "browser_version" #<UserAgent::Version 25.0.1364.152>>
<WME "user_agent" "platform" "Windows">
<WME "user_agent" "mobile" false>
<WME "params" "controller" "execute">
<WME "params" "action" "search_flights">
```

Quadro 39 – Triplas inicialmente geradas para a interface search_flights

Assim o acionamento desejado para a interface *search flights* só deverá ser feito caso a operação externa *search_flights* seja solicitada sem os parâmetros de que informam os códigos dos aeroportos (IATA) de partida e destino.

```
set{
  has "params", "action", "search_flights"
  neg "params", "airport_from", :_
  neg "params", "airport_to", :_
}
```

Quadro 40 – Regra para seleção da interface *search flights*

Para isso ocorrer, a regra definida faz uso dos dados dos parâmetros de navegação, conforme descrito no Quadro 40. A primeira condição define que a operação requerida (*action*) deve ser a *search_flights* e que os parâmetros *airport_from* e *airport_to* devem ser nulos, não existindo fatos (triplas)

correspondentes na base de conhecimento, conforme a segunda e terceira condição do quadro. O peso definido para essa interface é igual a um, apesar de não existirem nessa aplicação outras interfaces que conflitem com essa situação.

Composição abstrata

A composição abstrata da interface *search_flights* possui elementos para representar um cabeçalho com título, composição com rótulos (*label*) e campos para: aeroporto de partida e destino, data de partida e chegada, total de passageiros e botão para acionar a busca.

```
{name: "main_page", widget_type: "AbstractInterface", children:[
  {name: "header", widget_type: "CompositeInterfaceElement",
children: [
    {name: "title", widget_type: "ElementExhibitor"}] },
  {name: "content", widget_type: "CompositeInterfaceElement",
children:[
    {name: "label_from", widget_type: "ElementExhibitor"},
    {name: "airport_from", widget_type: "PredefiniteVariable"},
    {name: "label_to", widget_type: "ElementExhibitor"},
    {name: "airport_to", widget_type: " PredefiniteVariable "},
    {name: "label_depart", widget_type: "ElementExhibitor"},
    {name: "depart_date", widget_type: "IndefiniteVariable"},
    {name: "label_return", widget_type: "ElementExhibitor"},
    {name: "return_date", widget_type: "IndefiniteVariable"},
    {name: "label_pax", widget_type: "ElementExhibitor"},
    {name: "pax", widget_type: "PredefiniteVariable"},
    {name: "search_button", widget_type: "SimpleActivator"}
    ]}
  ]}
```

Quadro 41 – Composição abstratada interface *search flights*

A composição abstrata projetada pode ser vista no Quadro 41, com a sintaxe apresentada na seção 4.2.6.1.

Essa interface não prevê mudanças na composição abstrata em função dos dados. Assim, todos os elementos devem ser considerados em qualquer condição. Portanto nenhuma regra de seleção de elementos abstratos necessita ser declarada.

Mapeamento concreto

Para o mapeamento concreto da interface, a escolha dos *widgets* concretos levou em conta algumas funcionalidades e comportamentos desejados:

- A escolha do código dos aeroportos deveria ser feita com auto-sugestão, precisando o usuário digitar apenas as iniciais do local desejado e a operação *airports* retornaria as lista de aeroportos localizados;

- O aeroporto de destino não deveria ser o mesmo do de partida;
- A seleção da data de partida e retorno fosse através de um calendário;
- Os dados preenchidos fossem enviados para a operação *search_flights*.

Para atender as funcionalidades previstas foram definidas regras de mapeamento concreto conforme o Quadro 42. Cada regra foi numerada a fim de facilitar a identificação de cada uma.

```
1 maps_to abstract: "main_page", concrete_widget: "HTMLPage" ,
params: { title: "Search Fights", css_class: "search_page",
include_css: '/stylesheets/airline_tickets.css' }

2 maps_to  abstract: "header", concrete_widget: "HTMLComposition"

3 maps_to  abstract: "title", concrete_widget: "HTMLHeading",
params: {content: "Little Monk!"}

4 maps_to  abstract: "content", concrete_widget: "HTMLForm",
params: {css_class: "bounce-page", action:
"/execute/search_flights", method: "get"}

5 maps_to  abstract: "label_from", concrete_widget: "HTMLLabel",
params: {content: "From"}

6 maps_to abstract: "airport_from", concrete_widget:
"JQueryAjaxAutocomplete",
params: { json_source_url: "/execute/airports" }

7 maps_to  abstract: "label_to", concrete_widget: "HTMLLabel",
params: {content: "To"}

8 maps_to abstract: "airport_to", concrete_widget:
"JQueryAjaxAutocomplete",
params: { json_source_url: "/execute/airports",
params_from_elements: [ 'airport_from' ] }

9 maps_to  abstract: "label_depart", concrete_widget: "HTMLLabel",
params: {content: "Depart"}

10 maps_to  abstract: "depart_date", concrete_widget:
"JQueryDatePickerInput", params: { date_format: "d M, y",
min_date: 0 }

11 maps_to  abstract: "label_return", concrete_widget:
"HTMLLabel", params: {content: "Return"}

12 maps_to  abstract: "return_date", concrete_widget:
"JQueryDatePickerInput", params: { date_format: "d M, y",
min_date: 0 }

13 maps_to  abstract: "label_pax", concrete_widget: "HTMLLabel",
params: {content: "Pax"}

14 maps_to  abstract: "pax", concrete_widget: "HTMLFormSelect",
params: { options: ["1","2","3","4","5"] }
```

```
15 maps_to  abstract: "search_button", concrete_widget:
"HTMLFormButton", params: { css_class: "darkbookingbutton",
content: "Search"}
```

Quadro 42 – Regras de mapeamento concreto da interface *search flights*

Analisando as regras na ordem em que foram descritas no quadro acima, vale ressaltar alguns pontos de interesse:

- Nenhum elemento necessitou de condições para definir que *widget* concreto e parâmetros seriam utilizados. Sendo assim, todos os mapeamentos devem concretizados;
- A página de estilos utilizada na interface foi informada na regra 1, estando esse arquivo na pasta pública do Synth no caminho */stylesheets/airline_tickets.css*;
- O elemento abstrato *content* foi mapeado utilizando o *widget* concreto *HTMLForm* (regra 4). Para que os dados fornecidos sejam enviados para a operação externa *search_flights*, o parâmetro *action: "/execute/search_flights"* foi fornecido;
- O elemento *airport_from* foi mapeado com o *widget* concreto *JQueryAjaxAutocomplete* (regra 6), recebendo como parâmetro a URL para a operação *airports*, ou seja, */execute/airports*. Essa operação retorna um *Hash Json* com os aeroportos sugeridos. Esse *widget* insere uma caixa de texto e quando alguns caracteres são preenchidos a URL informada é requisitada e lista de opções é sugerida para seleção;
- O elemento *airport_to* (regra 8) utiliza exatamente o mesmo *widget* concreto do elemento *airport_from*, porém com um parâmetro adicional chamado *params_from_elements: ['airport_from']* que envia junto da requisição da URL fornecida, os dados de outros elementos da interface como parâmetros. Isso foi utilizado com o intuito da operação *airports* filtrar da lista retornada o aeroporto de partida;
- Os elementos *depart_date* e *return_date* foram mapeados para o *widget* concreto *JQueryDatePickerInput* (regras 10 e 12). Esse *widget* gera uma caixa de texto que ao receber foco exibe um

calendário. Para ambos elementos os parâmetros de mapeamento foram *date_format: "d M, y"*, *min_date: 0*. O primeiro parâmetro informa o formato da data e o segundo limita data mínima selecionada para o dia corrente;

Extensão concreta

Um comportamento adicional foi desejado para os campos de data da interface. Quando a data de partida (*depart_date*) for selecionada essa mesma data deve preencher o campo de data de retorno (*return_date*).

Para concretizar esse comportamento foi utilizada uma extensão concreta chamada *JQueryCopyTo*, conforme ilustrado no Quadro 43.

```
extend nodes: ['depart_date'], extension: 'JQueryCopyTo', params:
{ target: 'return_date'}
```

Quadro 43 – Extensão da interface *search flights*

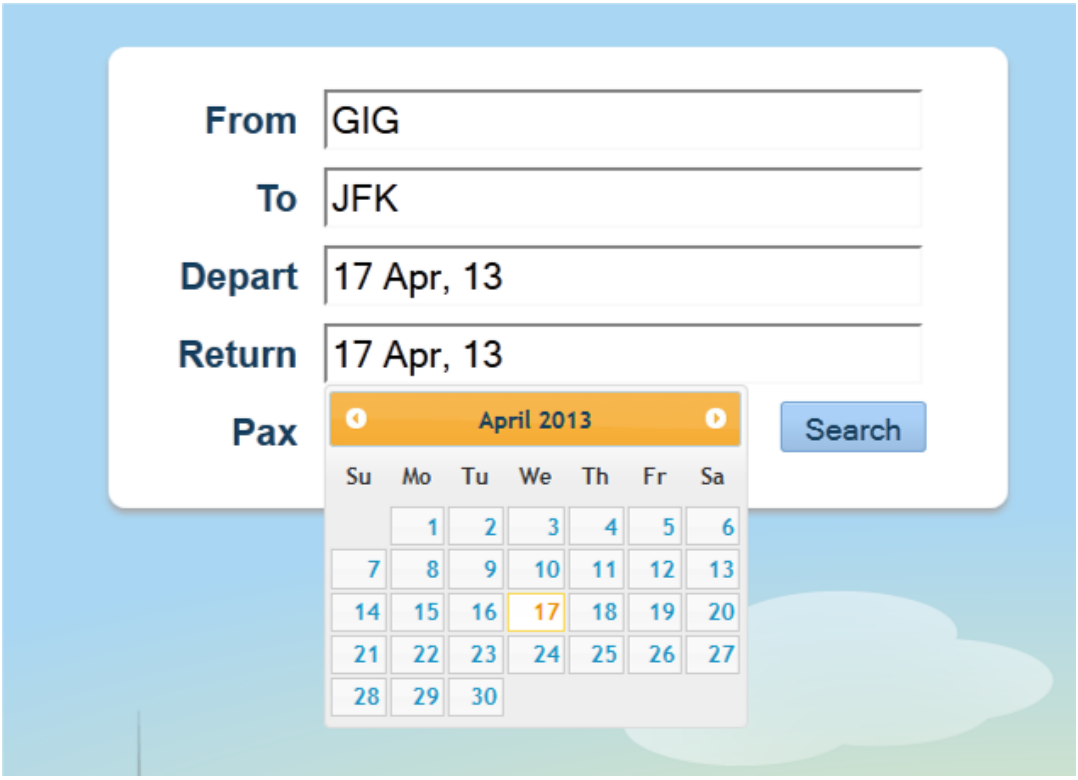

Figura 40 – Trecho da interface *search flights* - Calendário do *widget* concreto *JQueryDatePicker* e data copiada pela extensão *JQueryCopyTo*

5.4.2.
Flights IDX

A interface *Flights IDX* foi projetada para exibir um índice com os resultados da busca por voos realizada pela interface *Search Flights.*

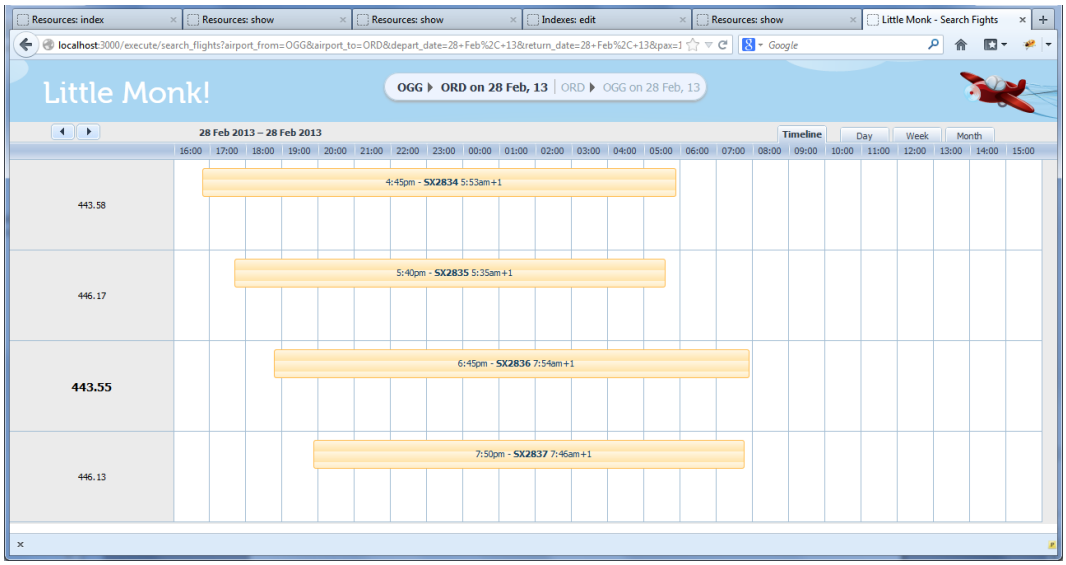

Figura 41 – Interface Flights IDX renderizada

A Figura 41 demonstra a interface resultante da busca por voos de *Kahului-Maui (HI US)* até *Chicago-O'Hare (IL US).* Essa interface utiliza uma apresentação em estilo *linha do tempo*, onde cada linha representa um voo localizado com seu preço, duração e identificação. O voo com o menor preço tem um destaque visual no seu valor. Também são exibidos no cabeçalho as datas e aeroportos.

Seleção da interface

A ativação da interface Flights IDX ocorre quando a operação externa *search_flights* invoca uma interface enviando um conjunto de fatos que contém os parâmetros de navegação e o índice instanciado gerando as triplas similares à do Quadro 44.

```
<WME "user_agent" "browser" "Chrome">
<WME "user_agent" "browser_version" #<UserAgent::Version 25.0.1364.152>>
<WME "user_agent" "platform" "Windows">
<WME "user_agent" "mobile" false>
<WME "params" "airport_from" "OGG">
<WME "params" "airport_to" "ORD">
<WME "params" "depart_date" "28 Mar, 13">
<WME "params" "return_date" "2 Apr, 13">
<WME "params" "pax" "1">
<WME "params" "search_button" "Search">
<WME "params" "controller" "execute">
<WME "params" "action" "search_flights">
```

```
<WME "navigational_element" "uri" "http://base#6a830-b654-b1e28">
<WME "http://base#6a830-b654-b1e28" "class" "SHDM::Index">
<WME "http://base#6a830-b654-b1e28" "index_title"  FlightsByAirportsIdx">
<WME "http://base#6a830-b654-b1e28" "index_name" "FlightsByAirportsIdx">
```

Quadro 44 – Triplas geradas para o índice de voos localizados

Para que a interface *Flights IDX* seja acionada, foi declarada uma regra de seleção que utiliza dados do índice navegacional para restringir a sua seleção, conforme o Quadro 45.

```
set{
  has "navigational_element", "uri", :URI
  has :URI, "class", "SHDM::Index"
  has :URI, "index_title", "FlightsByAirportsIdx"
  has "params", "airport from", :
  has "params", "airport_to", :_
}
```

Quadro 45 – Regra de seleção da interface Flights IDX

Essa regra contém condições para que a interface somente seja acionada quando o título do índice de navegação (propriedade *index_title*) for *FlightsByAirportsIdx* e quando os parâmetros *airport_from* e *airport_to* forem informados.

Composição abstrata

A composição abstrata da interface *Flights IDX* contempla adaptações na composição de seus elementos em duas situações: quando voos são localizados para serem exibidos e quando nenhum voo é localizado. Essas duas situações devem estar contempladas na descrição da composição abstrata. Para esse último caso, a interface gerada será como a da Figura 42.

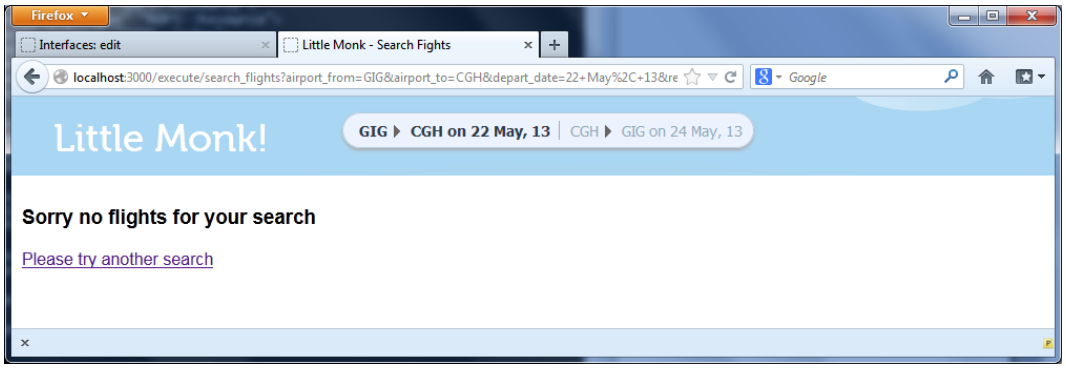

Figura 42 – Interface resultante para caso de nenhum voo for localizado

O Quadro 46 apresenta a descrição abstrata da interface *Flights IDX*. A interface contém composições para o cabeçalho da página (*header*) e para a parte de conteúdo (*content*) possuindo essa internamente duas composições: uma para a lista de voos (*flights*) e outra para o caso de nenhum não existirem voos para serem exibidos (*no_flights*).

```
{name: "main_page", widget_type: "AbstractInterface", children:[
  {name: "header", widget_type: "CompositeInterfaceElement", children: [
    {name: "title", widget_type: "SimpleActivator"},
    {name: "legs", widget_type: "CompositeInterfaceElement", children: [
      {name: "leg_go", widget_type: "CompositeInterfaceElement",
children:[
        {name: "leg_go_from", widget_type: "ElementExhibitor"},
        {name: "leg_go_depart", widget_type: "ElementExhibitor"},
        {name: "leg_go_arrive", widget_type: "ElementExhibitor"},
        ]},
      {name: "leg_back", widget_type: "CompositeInterfaceElement",
children:[
        {name: "leg_back_from", widget_type: "ElementExhibitor"},
        {name: "leg_back_arrive", widget_type: "ElementExhibitor"},
        {name: "leg_back_depart", widget_type: "ElementExhibitor"},
        ]}
      ]}

    ] },
  {name: "content", widget_type: "CompositeInterfaceElement", children:[
    {name: "flights", widget_type: "CompositeInterfaceElement",
repeatable: true, children:[
      {name: "flight", widget_type: "CompositeInterfaceElement",
children:[
        {name: "flight_price", widget_type: "ElementExhibitor"},
        {name: "flight_name", widget_type: " SimpleActivator"},
        {name: "flight_depart", widget_type: "ElementExhibitor"},
        {name: "flight_arrive", widget_type: "ElementExhibitor"}
        ]}
      ]},
    {name: "no_flights", abstract: "CompositeInterfaceElement",
children:[
      {name: "no_flights_msg", widget_type: "ElementExhibitor"},
      {name: "no_flights_anchor", widget_type: "SimpleActivator"}
      ]}
    ]}
  ]}
```

Quadro 46 – Descrição abstrata da interface Flights IDX

A composição *header* contém elementos para exibir o título da interface, os trechos de ida e volta e suas datas.

A composição *flights* foi projetada para exibir uma lista com os voos localizados na busca. Cada voo (*flight*) possui elementos para exibir o preço, a identificação do voo, hora de partida e chegada do voo.

Já a composição *no_flights* é composta apenas de um elemento para exibir uma mensagem e um ativador para navegar para outra operação.

Regras de seleção abstrata

Para realizar a seleção correta dos elementos no momento da renderização, duas regras foram descritas, conforme Quadro 47.

```
set "flights" do
  equal index.entries.empty?, false
end

set "no_flights" do
  equal index.entries.empty?, true
end
```

Quadro 47 – Regras de composição abstrata da interface Flights IDX

A regra para o elemento *flights* simplesmente é um teste que verifica se o índice possui entradas para serem exibidas e o elemento *no_flights* é exatamente a situação oposta quando a lista de entradas do índice é vazia.

Mapeamento concreto

O Quadro 48 apresenta as regras de mapeamento concreto declaradas para gerar a interface resultante.

```
1 maps_to abstract: "main_page", concrete_widget: "HTMLPage",
params: { id: 'index', title: "Little Monk - Search Fights", include_css:
'/stylesheets/airline_tickets.css'}

2 maps_to  abstract: "header", concrete_widget: "HTMLComposition"
3 maps_to  abstract: "title", concrete_widget: "HTMLAnchor",
params: {url: "/execute/search_flights", content: "Little Monk!"}

#== Legs
4 maps_to  abstract: "legs", concrete_widget: "HTMLComposition"

5 maps_to  abstract: "leg_go", concrete_widget: "HTMLComposition",
params: { css_class: "current_leg" }

6 maps_to  abstract: "leg_go_from", concrete_widget: "HTMLText",
params: {content: params[:airport_from] }

7 maps_to  abstract: "leg_go_depart", concrete_widget: "HTMLText",
params: {content: params[:airport_to] }

8 maps_to  abstract: "leg_go_arrive", concrete_widget: "HTMLText",
params: {content: params[:depart_date] }

9 maps_to  abstract: "leg_back", concrete_widget: "HTMLComposition"

10 maps_to  abstract: "leg_back_from", concrete_widget: "HTMLText",
params: {content: params[:airport_to] }

11 maps_to  abstract: "leg_back_arrive", concrete_widget: "HTMLText",
params: {content: params[:airport_from] }

12 maps_to  abstract: "leg_back_depart", concrete_widget: "HTMLText",
params: {content: params[:return_date] }

13 maps_to  abstract: "content", concrete_widget: "HTMLComposition"
```

```
#= Flights found
14 maps_to  abstract: "flights", concrete_widget:
"DHTMLXSchedulerComposition", params: {css_style:
"position:relative;width:100%;height:85%", skin: "glossy", collection:
index.entries, as: :flight}

15 maps_to  abstract: "flight", concrete_widget: "DHTMLXSchedulerEntry"

#= Best price!
16 maps_to  abstract: "flight_price", concrete_widget:
"DHTMLXSchedulerLineHead",
params: { css_class: "best_price", content: flight.price.label }do
   equal best_price, flight
end

#== Other prices
17 maps_to  abstract: "flight_price", concrete_widget:
"DHTMLXSchedulerLineHead", params: {content: flight.price.label }

#== Short Flights - The content is the flight number
18 maps_to  abstract: "flight_name", concrete_widget:
"DHTMLXSchedulerAnchorAjaxDialog",params: {content: flight.io::flight,
url: flight.context_anchor.target_url, dialog_id: "flight_info", title:
"Flight Info", width: 300 }do
  lte flight.duration.label.to_i, 3
end

#== Long Flights - The content is the text property
19 maps_to  abstract: "flight_name", concrete_widget:
"DHTMLXSchedulerAnchorAjaxDialog",params: {content: flight.text.label,
url: flight.context_anchor.target_url, dialog_id: "flight_info", title:
"Flight Info", width: 300 }

20 maps_to  abstract: "flight_depart", concrete_widget:
"DHTMLXSchedulerStartDate", params: {content: flight.start_date.label}

21 maps_to  abstract: "flight_arrive", concrete_widget:
"DHTMLXSchedulerEndDate", params: {content: flight.end_date.label}

#= No flights found
22 maps_to  abstract: "no_flights", concrete_widget: "HTMLComposition"

23 maps_to  abstract: "no_flights_msg", concrete_widget: "HTMLHeading",
params: {size: 3, content: "Sorry no flights for your search"}

24 maps_to  abstract: "no_flights_anchor", concrete_widget: "HTMLAnchor",
params: {url: "/execute/search_flights", content: "Please try another
search"}
```

Quadro 48 – Mapeamento concreto da interface Flights IDX

Segue abaixo uma lista de pontos considerados relevantes na descrição das regras de mapeamento da interface.

- Os parâmetros *airport_from*, *airport_to*, *depart_date* e *return_date* foram utilizados como valores para os dados do cabeçalho (regras 6 e 7);
- A composição do elemento *flights* utiliza o *widget* concreto *DHTMLXSchedulerComposition* (regra 14). Esse *widget* concreto é responsável por gerar a visualização em formato linha do tempo. Esse Widget concreto depende que os elementos filhos também utilizem os

Widgets da família *DHTMLXScheduler*. Cada elemento da coleção *index.entries* ficará visível pela variável *flight;*

- O elemento *flight_price* possui uma regra para aplicar um destaque caso o preço exibido seja o menor da lista (*css_class: "best_price"*). Caso não seja, outra regra efetua o mapeamento do elemento sem nenhuma alteração nos estilos (regras 16 e 17). A função para localizar o voo com o menor valor foi definida na operação *search_flights* e o elemento enviado como dado para interface;
- O elemento *flight_name* foi mapeado para o *widget* concreto *DHTMLXSchedulerAnchorAjaxDialog* que efetua uma requisição remota da url informada e o conteúdo recuperado é inserido em uma janela javascript - modal (regra 19). Nesse caso a url informada é do próprio nó navegacional em contexto e a interface desse contexto será apresentada na seção 5.4.3;
- Os elemento *flight_depart* e *flight_arrive* são mapeados para os *widgets* *DHTMLXSchedulerStartDate* e *DHTMLXSchedulerEndDate* (regra 20 e 21). Esses *widgets* recebem a data e hora do intervalo de início e fim de cada item exibido;
- As regras 22, 23 e 24 são referentes ao mapeamento do *widgets* necessários para renderizar a interface no caso de nenhum voo ser localizado. Elas geram basicamente elementos com a mensagem que nenhum voo foi localizado e também uma âncora de navegação para a interface de busca.

Extensões concretas

As extensões concretas aplicadas na interface *Flights IDX* tiveram a finalidade apenas de inserir elementos que não fazem parte da composição abstrata definida.

```
extend nodes: ['leg_go_from', 'leg_back_from'], extension: 'HTMLImage',
params: { content: "/images/airline-tickets/arrow_right.gif"}

extend nodes: ['leg_go_arrive', 'leg_back_depart'], extension:
'HTMLText', params: { content: " on ", insertion_position: "before"}
```

Quadro 49 – Extensões concretas aplicadas na interface Flights IDX

Todavia para um melhor resultado perceptível na interface final gerada e conforme descrição do Quadro 49 foram inseridas:

- Imagens com uma pequena seta após os elementos *leg_go_from* e *leg_back_from*;
- A *string* "on" antes dos elementos *leg_go_arrive* e *leg_back_depart*.

5.4.3. Flight Info

A última interface e a mais simples a ser apresentada da aplicação é a *Flight Info* ilustrada na Figura 43. Essa interface foi projetada para exibir informações adicionais sobre o voo como hora de partida e chegada, tipo de aeronave, número de escalas e distância em milhas.

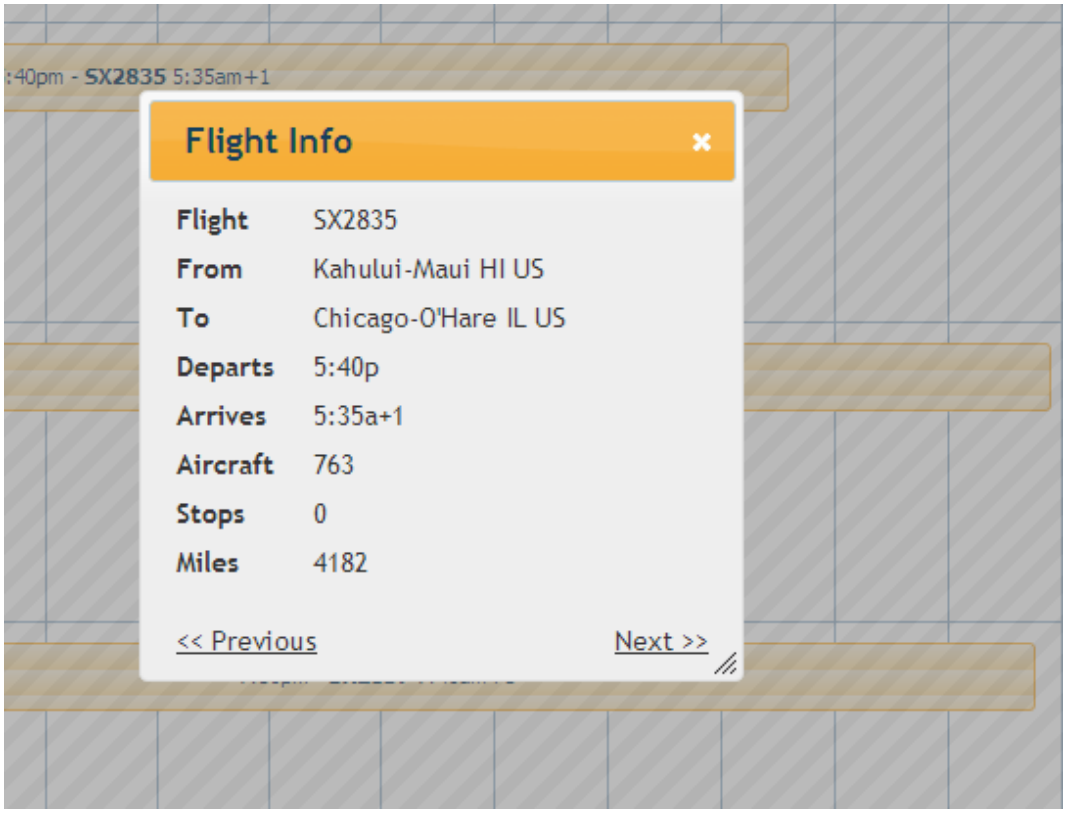

Figura 43 – Interface Flight Info com informações do voo

Seleção da interface

O acionamento da interface *Flight Info* é realizado pela operação externa context que envia como fatos dados do contexto, parâmetros e do nó em contexto.

Assim as triplas resultadas do conversor de triplas serão similares a do Quadro 50, podendo-se descrever regras esse valores.

```
<WME "params" "node" "http://www.snee.com/ns/flights#SX2836">
<WME "navigational_element" "uri" "http://base#69ab8750-4fda-11e2-b654-
00044b181e28">
<WME "http://base#69ab8750-4fda-11e2-b654-00044b181e28" "class"
"RDFS::Resource">
<WME "http://base#69ab8750-4fda-11e2-b654-00044b181e28" "class"
"SHDM::Context">
<WME "http://base#69ab8750-4fda-11e2-b654-00044b181e28" "context_name"
"FlightsByAirports">
<WME "http://base#69ab8750-4fda-11e2-b654-00044b181e28" "context_title"
"FlightsByAirports">
<WME "current_node" "uri" "http://www.snee.com/ns/flights#SX2836">
<WME "http://www.snee.com/ns/flights#SX2836" "class" "IO::Flight">
<WME "http://www.snee.com/ns/flights#SX2836" "rdf:type" "IO::Flight">
<WME "http://www.snee.com/ns/flights#SX2836" "fl:flightFromCityName"
"Kahului-Maui HI US">
<WME "http://www.snee.com/ns/flights#SX2836" "fl:flightToCityName"
"Chicago-O'Hare IL US">
<WME "http://www.snee.com/ns/flights#SX2836" "io:miles" "4182">
<WME "http://www.snee.com/ns/flights#SX2836" "io:depart" "6:45p">
<WME "http://www.snee.com/ns/flights#SX2836" "io:arrive" "7:54a+1">
<WME "http://www.snee.com/ns/flights#SX2836" "io:flight" "SX2836">
<WME "http://www.snee.com/ns/flights#SX2836" "io:aircraft" "763">
<WME "http://www.snee.com/ns/flights#SX2836" "fl:stops" "0, 6">
<WME "http://www.snee.com/ns/flights#SX2836" "io:meals" "D">
<WME "http://www.snee.com/ns/flights#SX2836" "io:duration" "8:09">
<WME "user_agent" "browser" "Chrome">
<WME "user_agent" "browser_version" #<UserAgent::Version 25.0.1364.152>>
<WME "user_agent" "platform" "Windows">
<WME "user_agent" "mobile" false>
<WME "params" "airport_from" "OGG">
<WME "params" "airport_to" "ORD">
<WME "params" "depart_date" "28 Mar, 13">
<WME "params" "return_date" "2 Apr, 13">
```

Quadro 50 – Triplas geradas para a interface Flight Info

Para efetuar a seleção da interface foi declarada a regra conforme o Quadro 51. Essa regra verifica se o nó navegacional é do tipo *SHDM::Context* e se o título do contexto instanciado é *FlightsByAirports*. Além disso é verificado se a identificação de algum nó foi enviada como parâmetro.

```
set{
  has "navigational_element", "uri", :NAV_ELEM_URI
  has :NAV_ELEM_URI, "class", "SHDM::Context"
  has :NAV_ELEM_URI, "context_title", "FlightsByAirports"
  has "params", "node", :_
}
```

Quadro 51 – Regra de seleção da interface Flight Info

Composição abstrata

A composição abstrata da interface Flight Info é apenas composta por elementos exibidores para os rótulos, dados do voo e âncoras de navegação entre os voos pertencentes ao mesmo contexto, conforme apresentado no Quadro 52.

```
{name: "flight", widget_type: "CompositeInterfaceElement",
children: [
  {name: "label_fl_number", widget_type: "ElementExhibitor" },
  {name: "fl number", widget type: "ElementExhibitor" },

  {name: "label_from", widget_type: "ElementExhibitor" },
  {name: "fl_from", widget_type: "ElementExhibitor" },

  {name: "label_to", widget_type: "ElementExhibitor" },
  {name: "fl_to", widget_type: "ElementExhibitor" },

  {name: "label_depart", widget_type: "ElementExhibitor" },
  {name: "fl_depart", widget_type: "ElementExhibitor" },

  {name: "label_arrive", widget_type: "ElementExhibitor" },
  {name: "fl_arrive", widget_type: "ElementExhibitor" },

  {name: "label_duration", widget_type: "ElementExhibitor" },
  {name: "fl_duration", widget_type: "ElementExhibitor" },

  {name: "label_aircraft", widget_type: "ElementExhibitor" },
  {name: "fl_aircraft", widget_type: "ElementExhibitor" },

  {name: "label_stops", widget_type: "ElementExhibitor" },
  {name: "fl_stops", widget_type: "ElementExhibitor" },

  {name: "label_miles", widget_type: "ElementExhibitor" },
  {name: "fl_miles", widget_type: "ElementExhibitor" },

  {name: "previous_flight", widget_type: "SimpleActivator" },
  {name: "next_flight", widget_type: "SimpleActivator" },
]}
```

Quadro 52 – Composição abstrata da interface Flight Info

Nenhuma regra de seleção de elementos abstratos foi necessária.

Mapeamento concreto

O Quadro 53 descreve todas as regras de mapeamento concreto definidas, onde em quase todos os elementos foram utilizados os *widgets* concretos *HTMLLabel* e *HTMLSpan*.

```
maps_to abstract: "flight", concrete_widget: "HTMLComposition"

maps_to abstract: "label_fl_number", concrete_widget: "HTMLLabel",
params: {content: "Flight"}
maps_to abstract: "fl_number", concrete_widget: "HTMLSpan", params:
{content: current_node.io::flight}

maps_to abstract: "label_from", concrete_widget: "HTMLLabel", params:
{content: "From"}
maps_to abstract: "fl_from", concrete_widget: "HTMLSpan", params:
{content: current_node.fl::flightFromCityName}

maps_to abstract: "label_to", concrete_widget: "HTMLLabel", params:
{content: "To"}
maps_to abstract: "fl_to", concrete_widget: "HTMLSpan", params: {content:
current_node.fl::flightToCityName}
```

```
maps_to abstract: "label_depart", concrete_widget: "HTMLLabel", params:
{content: "Departs"}
maps_to abstract: "fl_depart", concrete_widget: "HTMLSpan", params:
{content: current_node.io::depart}

maps_to abstract: "label_arrive", concrete_widget: "HTMLLabel", params:
{content: "Arrives"}
maps_to abstract: "fl_arrive", concrete_widget: "HTMLSpan", params:
{content: current_node.io::arrive}

maps_to abstract: "label_aircraft", concrete_widget: "HTMLLabel", params:
{content: "Aircraft"}
maps_to abstract: "fl_aircraft", concrete_widget: "HTMLSpan", params:
{content: current_node.io::aircraft }
maps_to abstract: "fl_aircraft", concrete_widget: "HTMLSpan", params:
{content: current_node.fl::plane }

maps_to abstract: "label_stops", concrete_widget: "HTMLLabel", params:
{content: "Stops"}
maps_to abstract: "fl_stops", concrete_widget: "HTMLSpan", params:
{content: current_node.fl::stops}do
  equal current_node.fl::stops.empty?, false
end

maps_to abstract: "fl_stops", concrete_widget: "HTMLSpan", params:
{content: "0"}

maps_to abstract: "label_miles", concrete_widget: "HTMLLabel", params:
{content: "Miles"}
maps_to abstract: "fl_miles", concrete_widget: "HTMLSpan", params:
{content: current_node.io::miles}

maps_to abstract: "previous_flight", concrete_widget: "JQueryAjaxAnchor",
  params: {content: "<< Previous", url:
current_node.previous_node_anchor.target_url, result_element_id:
"flight_info"}

maps_to abstract: "next_flight", concrete_widget: "JQueryAjaxAnchor",
params: {content: "Next >>", url:
current_node.next_node_anchor.target_url, result_element_id:
"flight_info"}
```

Quadro 53 – Regras de mapeamento concreto da interface Flight Info

Uma atenção especial só precisa ser dada para o elemento *fl_stops* que possui uma regra para verificar se o valor referente ao número de escalar for vazio o valor utilizado como conteúdo será zero (“0”). Outros dois elementos que valem atenção são o *next_flight* e *previous_flight* e que utilizam o *widget* concreto *JQueryAjaxAnchor* e recebem respectivamente como url a âncora para o próximo e anterior voos pertencentes ao mesmo contexto, efetuando uma requisição remota e retornando o resultado para a própria interface.

5.4.4.
Versões das interfaces para dispositivos móveis

Para exemplificar a geração de interfaces que se adaptam para dispositivos móveis, foram feitas apenas alterações pontuais nos modelos das interfaces **Search Flights** e **Flights IDX** já apresentadas anteriormente, não sendo necessárias alterações na interface **Flight Info**. Nos tópicos a seguir, só será dada ênfase nas inclusões realizadas em cada modelo.

5.4.4.1.
Search Flights

Conforme apresentado anteriormente, a interface *Search Flights* possui um formulário para inserção de informações para busca de voos. Na versão para dispositivos móveis a interface manteve a mesma composição abstrata, apenas ocorrendo alterações da etapa de mapeamento concreto.

```
#= Mobile
maps_to abstract: "main_page", concrete_widget: "HTMLPage" ,
params: { title: "Search Fights", css_class: "search_page",
include_css: '/stylesheets/airline_mob.css' }do
  has 'user_agent', 'mobile', true
end

#= Desktop
maps_to abstract: "main_page", concrete_widget: "HTMLPage" ,
params: { title: "Search Fights", css_class: "search_page",
include_css: '/stylesheets/airline_tickets.css' }

...

#= Mobile
maps_to  abstract: "pax", concrete_widget: "JQueryIncrementerFormInput",
params: {content: 1, min_value: 1}do
  has 'user_agent', 'mobile', true
end

#= Desktop
maps_to  abstract: "pax", concrete_widget: "HTMLFormSelect", params: {
options: ["1","2","3","4","5"] }
...
```

Quadro 54 – Novas regras de mapeamento concreto da interface Search Flights

Conforme ilustrado no Quadro 54, as únicas alterações realizadas foram nos mapeamentos:

- Do elemento **main_page** através da condição *<has 'user_agent', 'mobile', true>* que seleciona uma folha de estilo CSS diferente, caso ocorra uma requisição oriunda de um dispositivo móvel. Caso a condição seja diferente, a próxima regra utiliza o CSS definido originalmente;
- Do elemento **pax** para selecionar o número de passageiros. No caso de um dispositivo móvel, será utilizado o *widget* concreto *JQueryIncrementerFormInput* que renderiza um seletor numérico incremental, que faz o papel da caixa de seleção da interface padrão.

A Figura 44 apresenta o resultado da renderização da interface com destaque para o seletor de total de passageiros, com botões para incrementar e decrementar o valor desse elemento. Essa decisão de adaptação foi com a intenção de facilitar a seleção do valor em dispositivos com telas de toque.

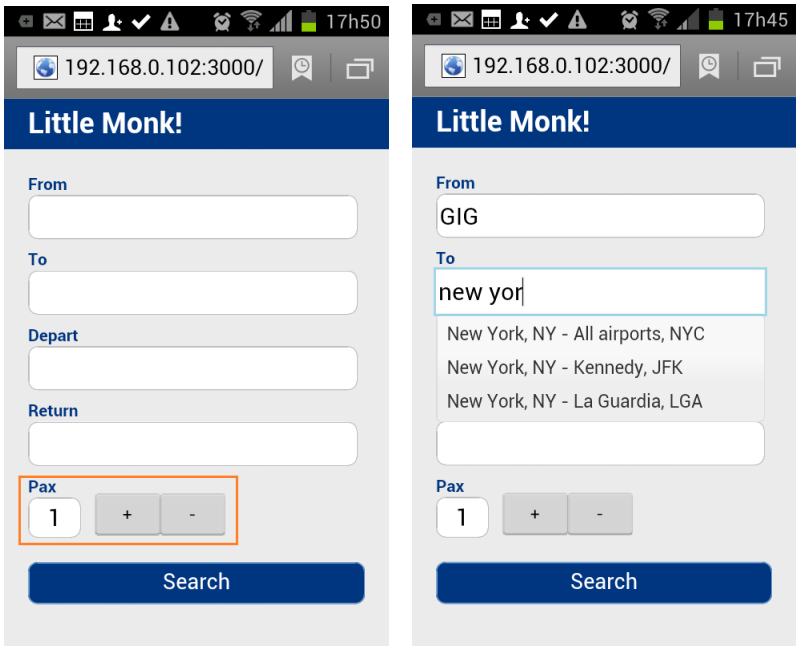

Figura 44 – Interface Search Flights com o Widget JQueryIncrementerFormInput

5.4.4.2. Flights IDX

A interface *Flights IDX* apresenta um índice com a lista de voos localizados para uma determinada origem e destinos selecionados. Na interface original, essa lista é apresentada através de uma linha de tempo (conforme Figura 41), porém os

dispositivos móveis normalmente tem uma limitação de espaço de exibição, devido o tamanho reduzido das telas. Por essa razão, algumas adaptações foram realizadas na composição abstrata e também no mapeamento concreto.

Composição abstrata

Para adaptar a interface *Flights IDX* para dispositivos móveis, um novo elemento foi inserido na composição abstrata e alguns elementos foram inibidos através das regras de seleção abstrata.

```
...
{name: "leg_back", widget_type: "CompositeInterfaceElement", children:[
        {name: "leg_back_from", widget_type: "ElementExhibitor"},
        {name: "leg_back_arrive", widget_type: "ElementExhibitor"},
        {name: "leg_back_depart", widget_type: "ElementExhibitor"},
        ]},
      {name: "next_leg", widget_type: "SimpleActivator"}
      ]},
    ] },
...
```

Quadro 55 – Trecho da composição abstrata de Flights IDX que recebeu um novo elemento

O Quadro 55 apresenta o trecho da composição abstrata onde o elemento *next_leg* foi inserido, com o objetivo de inserir uma âncora de navegação entre a lista de voos de ida e volta[61].

```
#= Desktop
set "leg_back" do
  has "user_agent", "mobile", false
end

#= Mobile
set "next_leg" do
  has "user_agent", "mobile", true
end

...
```

Quadro 56 – Regras de seleção abstrata da interface Flights IDX

[61] Essa navegação é apenas hipotética não existindo na aplicação de exemplo.

Já o elemento **leg_back** foi inibido para dispositivos móveis para não inserir as informações do voo de volta no cabeçalho da página, dando lugar ao elemento **next_leg**, conforme regras do Quadro 56.

Mapeamento concreto

O mapeamento concreto da interface *Flights IDX* utilizou tanto alterações de parâmetros dos *widgets* concretos utilizados na interface original, quando novos mapeamentos para *widgets* concretos específicos.

```
#= MOBILE
maps_to abstract: "main_page", concrete_widget: "HTMLPage" , params: {
id: 'index', title: "Little Monk - Search Fights", include_css:
'/stylesheets/airline_mob.css'}do
  has "user_agent", "mobile", true
end

#= Desktop
maps_to abstract: "main_page", concrete_widget: "HTMLPage" , params: {
id: 'index', title: "Little Monk - Search Fights", include_css:
'/stylesheets/airline_tickets.css'}

...

#= MOBILE MAPS

maps_to  abstract: "next_leg", concrete_widget: "HTMLAnchor", params:
{url: "?next=1", content: "Return Flight >"}

#= Flights found
maps_to  abstract: "flights", concrete_widget: "HTMLComposition",
params: {collection: index.entries, as: :flight}do
  has "user_agent", "mobile", true
end

maps_to  abstract: "flight", concrete_widget: "HTMLComposition", params:
{css_class: "flight"}do
  has "user_agent", "mobile", true
end

maps_to  abstract: "flight_price", concrete_widget: "HTMLSpan", params:
{content: "$ #{flight.price.label}", css_class: "price" }do
  has "user_agent", "mobile", true
end

maps_to  abstract: "flight_name", concrete_widget:
"JQueryAjaxAnchorDialog",
params: {content: flight.io::flight, css_class: "flight_name" , url:
flight.context_anchor.target_url, dialog_id: "flight_info", title:
"Flight Info", width: 300 }do
  has "user_agent", "mobile", true
end

maps_to  abstract: "flight_depart", concrete_widget: "HTMLSpan", params:
{content: flight.io::depart, css_class: "flight_depart" }do
  has "user_agent", "mobile", true
end
maps_to  abstract: "flight_arrive", concrete_widget: "HTMLSpan", params:
{content: flight.io::arrive, css_class: "flight_arrive"}do
```

```
  has "user_agent", "mobile", true
end
```

Quadro 57 – Mapeamento concreto da interface Flights IDX para dispositivos móveis

O Quadro 57 apresenta as inclusões no mapeamento concreto originalmente proposto para realizar adaptação para dispositivos móveis (DM), sendo cada ponto comentado a seguir:

- O elemento **main_page** recebeu uma nova configuração de parâmetro para diferenciar a folha de estilos CSS utilizada no caso da interface ser requerida por um DM;
- O elemento **next_leg** foi mapeado com o *widget* concreto **HTMLAnchor**, sendo somente acionado para DM;
- O elemento **flights** e **flight** utilizam o *widget* concreto **HTMLComposition** no lugar dos da família de *widgets* **DHTMLXScheduler**;
- Os elementos **flight_price**, **flight_depart** e **flight_arrive** passaram a utilizar o widget **HTMLSpan**;
- O elemento **flight_name** foi mapeado para o *widget* concreto **JQueryAjaxAnchorDialog** que efetua uma requisição Ajax de uma URL e apresenta o resultado em uma janela javascript - modal;
- Os elementos **flight_depart** e **flight_arrive** utilizam respectivamente como parâmetro de conteúdo (*content*) os valores das propriedades **flight.io::depart** e **flight.io::arrive**;
- Nenhuma extensão concreta foi incluída.

O resultado da renderização pode ser visto na Figura 45, onde o resultado dos voos localizados é apresentado em uma lista. As informações do voo selecionado são exibidas em uma janela javascript - modal.

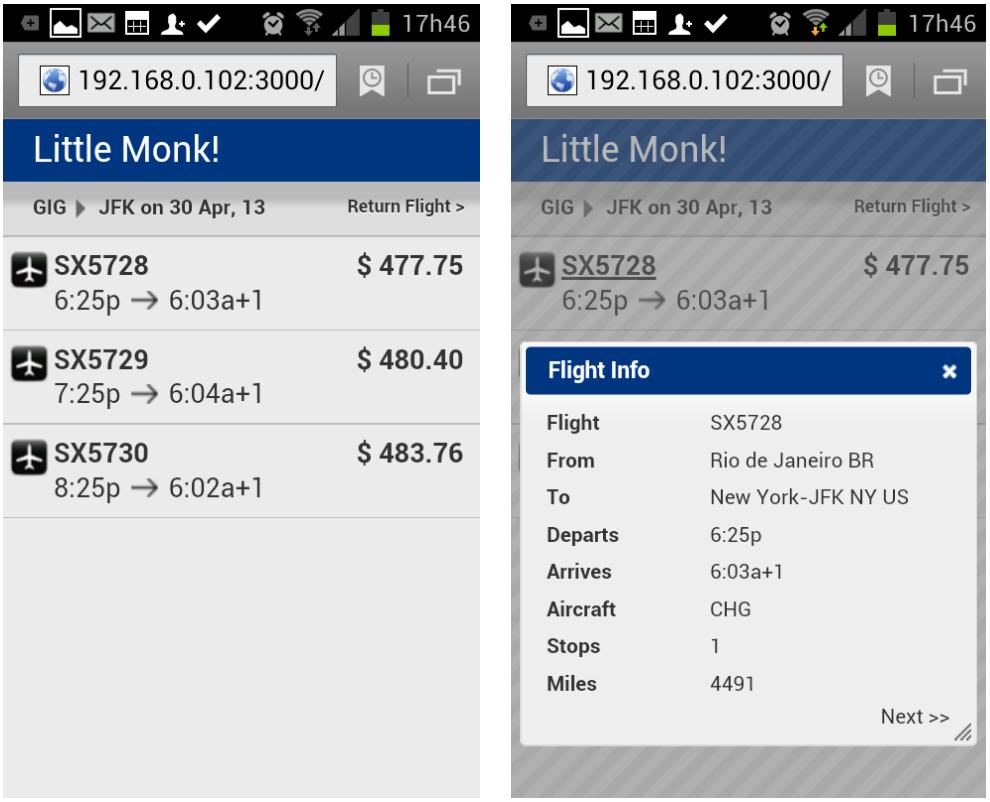

Figura 45 – Resultado da renderização da interface Flights IDX

Diversas outras opções de adaptação das interfaces para dispositivos móveis poderiam ser adotadas, inclusive a de modelar uma interface especialista para tratar esse tipo de requisição. Contudo, cabe ao projetista escolher qual a melhor estratégia de modelagem para cada interface projetada.

6
Conclusão

6.1.
Trabalhos relacionados

Esse trabalho está relacionado com uma série de abordagens para o desenvolvimento de aplicações e interfaces dirigidas por modelos ou MBUI's, como por exemplo: as etapas do Cameleon Framework Reference, os modelos de interface do SHDM, a UsiXML, UIML, MARIA / TERESA e outras abordagens comentadas no capítulo 2. Todavia, ocorrem discordâncias em alguns aspectos como, por exemplo, o não uso de notação XML na especificação abstrata e concreta e a possibilidade de se tratar de questões de dependência de plataforma e dispostivos no modelo concreto. Em abordagens como UsiXML e MARIA, o modelo concreto não trata questões de dependência com dispositivos e plataformas. Essas questões são tratadas em uma etapa final de geração de interfaces. Outra diferença é que esse trabalho está restrito à geração de interfaces de aplicações de internet, sem a pretensão de gerar interfaces multi-modais.

Trabalhos específicos de comparações entre UIDL's existentes já foram realizados, como o de Pohja [2011] e o de Garcia et al. [2009]. Esse último em especial realiza a comparação das UIDL's levando em conta três aspectos:

- O perfil de camadas aderentes às etapas de transformação do Cameleon Reference Framework (CRF);
- Um conjunto de dimensões para as características gerais das UIDL's (especificidade, disponibilidade, tipo de uso, instituição e grau de uso);
- As propriedades das diferentes UIDL`s segundo um conjunto de oito critérios (apresentados mais adiante).

Sendo assim, é possível incluir o presente trabalho na comparação das UIDL's feitas por Garcia et al. [2009] utilizando os aspectos citados anteriormente[62].

[62] Para uma melhor percepção e uma possível comparação entre as demais UIDL`s avaliadas por Garcia et al. [2009] é recomentada a leitura de tal artigo a priori.

O primeiro aspecto pode ser visto através da Figura 46 que apresenta os modelos e artefatos existentes no método desse trabalho distribuído entre as camadas previstas no CRF.

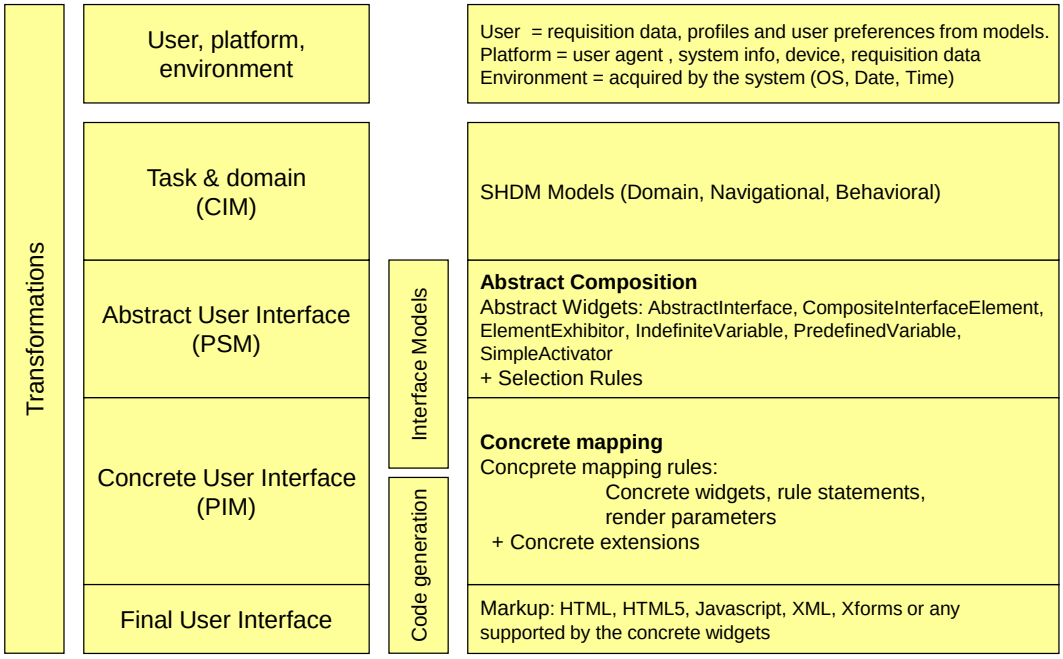

Figura 46 - Modelo de interfaces prosposto segundo o CRF

O segundo aspecto leva em conta algumas características gerais onde cada uma recebe valores conforme os dados a seguir:

- **Especificidade**: indica se o UIDL poderia ser usado em uma ou múltiplas plataformas ou dispositivos;
- **Disponibilidade**: 0 = nenhuma informação disponível, 1 = não disponível, 2 = pouco disponível, 3 = moderadamente disponível, 4 = completamente disponível e 5 = completamente disponível com meta-modelos;
- **Tipo de uso**: informa se a UIDL é utilizada em pesquisa ou na indústria;
- **Instituição**: qual organização está por trás do desenvolvimento;
- **Grau de uso**: 0 = desconhecido, 1 = uma pessoa, 2 = duas ou mais pessoas, 3 = uma organização, 4 = duas ou mais organizações e 5 = ampla utilização.

Com base nisso foi gerada a Tabela 2 que dá valores para características apresentadas para o método desse trabalho e também faz a reprodução dos valores estipulados (pelo autor da pesquisa original) para TeresaXML e a USIXML.

UIDL	Especificidade	Disponibil.	Tipo de uso	Instituição	Grau de uso
Teresa XML	Multiplataforma e multidispositivo	4	Pesquisa	HCI Group of ISTI-C.N.R.	3
USIXML	Multiplataforma	5	Pesquisa	UCL	3
SHDM Interface Models (presente trabalho)	Multiplataforma e multidispositivo (web applications)	4	Pesquisa	PUC-Rio	2

Tabela 2 - Comparação entre algumas UIDLs segundo características gerais

O terceiro e último aspecto analisa as UIDL's através de algumas propriedades utilizando um conjunto de oito critérios. Cada um deles recebe valores conforme segue abaixo:

- **Modelos disponíveis**: define os aspectos da interface que podem ser especificados na descrição das interfaces com o usuário (UI's). Os modelos disponíveis para comparação são: modelo de *tarefa*, modelo *domínio*, modelo de *apresentação* e o modelo de *diálogo*;
- **Metodologia**: diferentes abordagens para especificar as UI's, onde podem ser definidos como: 1*) Especificação de uma UI* para cada um dos diferentes contextos de utilização. 2) *Especificação genérica de UI*, válida para todos os diferentes contextos de uso;
- **Ferramentas**: ferramentas que suportam alguma linguagem que auxilia o projetista na descrição da UI e renderiza a especificação para outra linguagem e / ou plataforma;
- **Linguagens suportadas**: linguagens de programação em que a descrição das interfaces podem ser traduzidas;
- **Plataformas suportadas**: plataformas computacionais em que a linguagem pode ser executada e/ ou interpretada;
- **Nível de abstração**: cada UIDL pode apresentar a capacidade de expressar uma interface executável (*nível de instância*), um ou mais modelos envolvidos no desenvolvimento da UI (*nível de modelo*), como esses modelos são construídos (*nível de meta-modelo*) e quais são os conceitos fundamentais em que essa operação se baseia (*nível de metameta-modelo*);

- **Quantidade de *tags***: para atingir o nível anterior de abstração, cada UIDL manipula certa quantidade de *tags*, o que é também altamente dependente da cobertura dos conceitos;
- **Cobertura dos conceitos**: dependendo do nível de abstração, cada UIDL pode introduzir algum conceito específico x conceitos genéricos.

Por fim, utilizando os critérios listados anteriormente foi construída a Tabela 3 onde foram preenchidos os valores para cada critério definido em relação a este trabalho. Para fins de comparação também foram reproduzidos os valores para a UIDL TeresaXML.

UIDL	Modelo	Metodologia	Ferramentas	Linguagens suportadas	Plataformas suportadas	Nível de abstração	Quantidade de tags	Cobert. dos conceitos
TeresaXML	Apresentação, tarefa e diálogo	Especificação genérica de UI	CTTE Tool for task Models Teresa	Markup: Digital TV, VoiceXML, XHTML/SVG, X+V Programação: C#	DigitalTV, Mobile, Desktop PC	Nível de modelo	19 tags	Mappings, models, platform, task, input, output
SHDM Interface Models (presente trabalho)	Domínio, Apresentação e diálogo	Especificação de uma UI ou Especificação genérica de UI	Ambiente de autoria da ferramenta Synth	Markup: HTML, HTML5, Javascript, XML, json Programação: Ruby	Aplicações web (desktop, mobile, etc)	Nível de modelo e instância	10 tags (ignorando parâmetros dos widgets e regras)	Selection Interface Rule, Abstract Widgets, element selection Rules, Concrete mapping rules, Concrete extensions

Tabela 3 - Tabela com critérios para avaliação das UIDLs

As tabelas originais com os demais valores para outras UIDL's podem ser consultadas no Apêndice III.

6.2. Avaliações gerais

A inclusão de uma abordagem que avalia os dados sobre um conjunto de regras em todas as etapas da modelagem das interfaces e é capaz de gerar interfaces em tempo de execução a partir dessa avalição possibilita que métodos como o SHDM possam fazer uso de informações do meta-modelo para compor as interfaces adequadamente[63]. Sendo assim, os tipos e propriedades dos dados (metadados) também serão levados em consideração na composição das regras e

[63] Um exemplo breve do uso de informações do meta-modelo de uma aplicação SHDM é o da interface do exemplo 5.4.2 que utiliza informações oriundas do modelo navegacional para ajustar a composição da interface.

do mapeamento concreto, que podem avaliar se determinado elemento deve ser selecionado ou mapeado concretamente em função desses metadados. A avaliação sobre dados e metadados é uma importante diferença da abordagem aqui apresentada em relação às anteriores adotadas no SHDM. Sem esta possiblidade, o desenvolvedor era obrigado a codificar concretamente todas as interfaces da aplicação, a fim de atender certos requisitos e comportamentos desejados, e com isso, os modelos de interface existentes eram pouco utilizados, pois os projetistas se contentavam com as interfaces padrão geradas automaticamente.

Outro aspecto a ser considerado é a geração de interfaces que atendam às mudanças de dispositivos. Atualmente existem versões de aplicativos e interfaces para computadores tradicionais, *smartphones* e *tablets*. O que se observou é que o desenvolvedor deve optar entre três estratégias de implementação ao projetar interfaces para esse propósito:

- Projetar distintas interfaces para cada dispositivo;
- Projetar uma mesma interface que possua diferentes composições em função do dispositivo. Por exemplo, propagandas (*banners*) são suprimidas em dispositivos móveis;
- Manter a composição da interface, porém os *widgets* concretos ou as configurações de mapeamento são especializados por dispositivo. Por exemplo, o uso diferentes tabelas de estilos CSS para cada dispositivo ou a alteração do valor textual de um *widget* acionador de ‘Clique para efetuar a compra’ para simplesmente ‘Comprar’ num dispositivo móvel.

Qualquer uma das alternativas citadas, ou suas combinações, são suportadas pelo método e arquitetura aqui apresentados, ficando a cargo do projetista decidir qual melhor lhe atende.

Apesar não terem sido efetuados testes robustos de desempenho, numa avaliação preliminar, a utilização de uma máquina de regras de produção e um interpretador de interfaces não geraram impacto significativo de desempenho se comparado ao tempo gasto com as consultas na camada de persistência do ambiente. Todavia, cabem testes mais elaborados para quantificar o impacto da avaliação de regras e interpretação no tempo total de renderização das interfaces.

O último ponto a ser discutido é a questão da plasticidade das interfaces. Thevenin e Coutaz [1999] introduziram o conceito de plasticidade como a

capacidade de uma interface em suportar variações de características físicas do sistema (plataforma) e do meio ambiente, preservando a usabilidade.

Originalmente (Figura 47), quatro eixos foram propostos para cobrir as possíveis mudanças nas características que uma interface é exposta: contexto de uso (*target*), significados da adaptação (*means*), adaptação dinâmica ou estática (*time*) e que atores responderão às adaptações (*actor*).

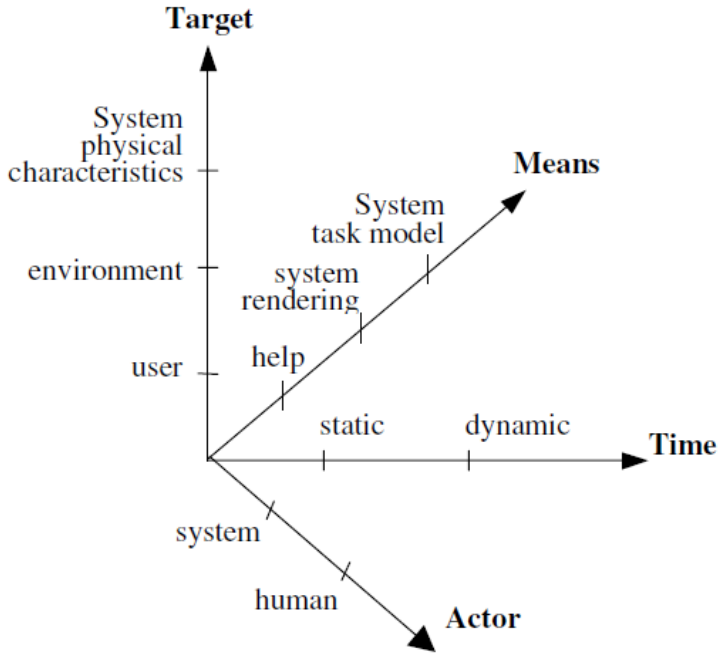

Figura 47 – Eixos e aspectos previstos sob a perspectiva da plasticidade

Mesmo sem uma análise profunda, é possível perceber que boa parte dos aspectos previstos na plasticidade das interfaces pode ser considerada utilizando a arquitetura aqui proposta:

- As mudanças de contexto podem ser percebidas através dos dados transmitidos à interface e convertidos em fatos para as regras. Assim, o projetista pode prever as situações de adaptação em função de alterações no perfil do usuário corrente, nos valores capturados no ambiente e na plataforma utilizada;
- Mudanças no mecanismo de renderização e resposta a alterações no modelo de tarefas podem ser feitas através de regras que troquem ou reconfigurem os *widgets* concretos;
- As adaptações podem ocorrer de forma estática, ou seja, quando uma requisição é realizada e a interface é renderizada com a adaptação

prevista. Ou ainda dinâmica, quando componentes da interface são requeridos via requisição remota, transmitindo a cada requisição valores que podem ser percebidos como mudanças em um dos eixos.

Posteriormente, Coutaz e Calvary [2012] aprofundaram a discussão sobre plasticidade definindo o que foi chamado de "O problema de espaço da plasticidade das interfaces", afirmando que a adaptação[64] de uma interface deve ser por **remodelagem** ou **redistribuição**, adicionando novos eixos de perspectivas. A análise dessas novas perspectivas fica como recomendação para trabalhos futuros.

6.1. Contribuições

As principais contribuições desse trabalho foram o método e a arquitetura propostos, que juntos oferecem às interfaces suporte à adaptações decorrentes de mudanças no contexto de uso, tornando-as mais sensíveis aos dados.

Adicionalmente, outras contribuições também são relevantes:

- Oferecer ao SHDM modelos de interfaces mais flexíveis e aderentes aos conceitos e características encontradas em outras UIDL's modernas;
- Possibilitar que as interfaces que usam dados da Web Semântica (RDF) possam se compor a partir da avaliação dos metadados sobre os recursos a serem exibidos;
- Um ambiente de autoria e execução das interfaces, integrado ao Synth;
- Um caso concreto de aplicação de regras de produção para construção de interfaces, integrado aos modelos de domínio de uma aplicação;
- Uma arquitetura para construção, interpretação e distribuição de *widgets* concretos;

[64] Conceitos de interfaces adaptativas não devem ser confundidos com os de interfaces adaptáveis, não sendo esse aspecto explorado nesse trabalho.

- Um conjunto de *widgets* e extensões concretas para os casos mais comuns de interfaces de aplicações;
- Uma especificação padronizada das capacidades configuráveis dos *widgets* concretos através do arquivo de manifesto.

6.2. Trabalhos futuros

Como trabalhos futuros algumas avaliações são sugeridas para o método:

- Avaliar quais são os casos de adaptação de interfaces são mais freqüentes e que práticas podem ser adotadas para cada caso verificado;
- As extensões concretas apresentadas nessa proposta não fazem uso de regras para serem consideradas na interface final renderizada, sendo dependentes dos *widgets* concretos mapeados. Porém, um levantamento de casos de uso mais complexos pode indicar se também serão necessárias regras para as extensões;
- Todos os efeitos, respostas a eventos, animações e transições foram encapsulados nas funcionalidades dos widgets concretos que devem ser abrangentes o suficiente para cobrir essas situações. Porém podem ser desejados que esses aspectos possam ser modelados mais abstratamente, cabendo análise nesse aspecto;
- Verificar que outros ambientes podem utilizar o método e a arquitetura implementada;
- Efetuar testes controlados de usabilidade do método, propondo exercícios que atendam a casos de uso predeterminados com o intuito de detectar dificuldades na assimilação do método. Avaliar também o ambiente de autoria, a legibilidade das regras de seleção e mapeamento concreto.

Para o ambiente de autoria e arquitetura de implementação são recomendadas certas melhorias:

- Projetar uma interface de autoria gráfica para composição abstrata e para declaração das regras das interfaces;

- Mecanismos mais robustos de validação de sintaxe da composição abstrata;
- No ambiente de autoria do Synth, desenvolver um mecanismo gerenciador de arquivos relacionados à aplicação e às interfaces, como folha de estilos CSS, imagens e documentos;
- A construção de novas famílias de *widgets* concretos que ofereçam ao projetista suporte para as mais diversas funcionalidades, além de famílias de *widgets* concretos especializados para determinados dispositivos, plataformas e exibição de dados de naturezas específicas (por exemplo, *widgets* para JQueryMobile[65], iUI-JS[66], Foundation[67], etc);
- Avaliar, com casos mais complexos, se o mecanismo de resolução de dependência entre *widgets* da mesma interface tem a robustez necessária.

[65] jquerymobile.com

[66] http://www.iui-js.org/

[67] http://foundation.zurb.com

7
Referências bibliográficas

BERTI S.; CORREANI F.; PATERNÒ F.; SANTORO C. **The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels**, ISTI-CNR,Via G.Moruzzi 1 ,Pisa, Italy. Leveles, Proceedings Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, p. 103-110, 2004.

BOMFIM, M. H. S. **Um método e um ambiente para o desenvolvimento de aplicações na Web Semântica.** Dissertação de Mestrado, PUC-Rio em 2011.

BOZZON, A. et al. **Conceptual Modeling and code Generation for Rich Internet Applications**. In: 6TH International Conference on Web Engineering – ICWE '06, 2006, Palo Alto. Proceedings of the 6th international conference on Web engineering. New York: ACM Press, 2006. p. 353–360.

CALVARY, G., COUTAZ, J., BOUILLON, L., FLORINS, M., LIMBOURG, Q., MARUCCI, L., PATERNÒ, F., SANTORO, C., SOUCHON, N., THEVENIN, D., VANDERDONCKT, J., **The CAMELEON Reference Framework**, Deliverable 1.1, CAMELEON Project, 2002. Disponível em: <http://giove.isti.cnr.it/projects/cameleon/pdf/CAMELEON%20D1.1RefFramework.pdf>. Acesso em 10/10/2012

CERI, S.; FRATERNALI, P.; BONGIO, A. **Web Modeling Language (WebML): a modeling language for designing Web sites**. Procs of the WWW9 Conf., Amsterdam. Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking, May 2000.

CORMEN, T.;LEISERSON, C.;RIVEST,R.;STEIN, C. **Algoritmos - Teoria e Prática**. Segunda edição, Editora Campus, 2002.

COUTAZ, J., CALVARY, G. **HCI and Software Engineering for User Interface Plasticity. In Human Computer Handbook: Fundamentals, Evolving technologies, and Emerging Applications**, 3rd edition, Julie Jacko Ed., Taylor and Francis Group Ltd Publ., May 2012.

DAVIS, R., SHROBE, H., SZOLOVITS, P. **What Is a Knowledge Representation?**, AI Magazine Volume 14 Number 1, 1993

DOORENBOS, R. B. **Production Matching for Large Learning Systems.** CMU-CS-95-113. Computer Science Department. Carnegie Mellon University. January 31, 1995

FORGY, C. **On the efficient implementation of production systems**. Ph.D. Thesis, Carnegie-Mellon University, 1979.

FOWLER, M. **Rules Engine**. 2009. Disponível em <http://martinfowler.com/bliki/RulesEngine.html>. Acesso em 10/12/2012.

GARCIA, P. N. F. **Module 5 Project - Using Jena to manage RDF data**. OCW - UC3M. 2010, July 27, Disponível em <http://ocw.uc3m.es/ingenieria-telematica/web-2.0-and-web-3.0-technologies/rdfmngt/module-5-project>. Acesso em 27/12/2012.

GARCIA J. G.; CALLEROS J. M. G.; VANDERDONCKT J. **A theoretical survey of user interface description languages** : Preliminary results. Web Congress, 2009. LA-WEB '09. Latin American, Merida, Yucatan, p 36-43.

GUPTA, A.; FORGY C. L.; **Measurements on Production Systems.** Computer Science Department, Carnegie Mellon University, Paper 1472, December 1983.

HAY, D.;KOLBER, A.; HEALY, K. A.; HALL, J. **Defining Business Rules ~ What Are They Really?** The Business Rules Group, the GUIDE Business Rules Project, Final Report, revision 1.3, July, 2000.

HELMS, J., SCHAEFER, R., LUYTEN, K., VERMEULEN, J., ABRAMS, M., COYETTE, A., VANDERDONCKT, J., **Human-Centered Engineering with the User Interface Markup Language**, Human-Centered Software Engineering, Chap. 7, Springer Verlag, London, 2009 pp 141-173

HOOVER, S.P.; RINDERLE, J.R.; FINGER, S. **Models and abstractions in design.** Design Studies, 1991. p. 237-245.

KLEINBERG, J.; TARDOS E. **Algorithm Design**. Pearson Education, 2006

KOCH, N.; KNAPP, A.; ZHANG, G; AND BAUMEISTER, H.; **UML-based Web Engineering: An Approach based on Standards (book chapter)**. In Web Engineering: Modelling and Implementing Web Applications. Gustavo Rossi, Oscar Pastor, Daniel Schwabe and Luis Olsina (Eds.), chapter 7, 157-191, ©Springer, HCI, 2008.

LAVIE, T.; MEYER, J.; **Benefits and costs of adaptive user interfaces**, Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer Sheva, Israel, January 2010.

LIMA, F. **Modelagem semântica de aplicações na WWW**, Tese de Doutorado, PUC-Rio em 2003.

LIMBOURG Q., VANDERDONCKT J., MICHOTTE B., BOUILLON L., LÓPEZ-JAQUERO V., **USIXML: A Language Supporting Multi-path Development of User Interfaces**. Working Conference on Engineering for Human-Computer Interaction and International Workshop on Design Specification and Verification of Interactive Systems(EHCI/DS-VIS), Hamburg, Germany , 200-220, 2004

LUNA, A. **Geração de Interfaces RIA Dirigida por Ontologias**. Dissertação de Mestrado, PUC-Rio em 2009.

MEIXNER, G.; PATERNÓ, F.; VANDERDONCKT, J. **Past, Present, and Future of Model-Based User Interface Development**. i-com, 10, 3 (2011), 2-11.

MENDES S. A. **FlexiXML - Um animador de modelos de interfaces com o utilizador**, 2009

MYERS, B.; ROSSON, M. B.: **Survey on User Interface Programming**. Proc. of the 10th Annual CHI Conference on Human Factors in Computing Systems, pp. 195-202, 1992.

MODEL-BASED UI WORKING GROUP. **Introduction to Model-Based User Interfaces**, Working Draft, 2012, http://www.w3.org/wiki/Model-Based_ User_Interfaces, Disponível em https://docs.google.com/document/d/1Xp50GZ8E fY017AT_pCMBq5PeK8cwNEZi1a8hXexJkCc/edit#heading=h.cdg7frhj8dlu, Acessado em 31/01/2013.

MOURA, S. S. **Desenvolvimento de Interfaces Governadas por Ontologias para Aplicações na Web Semântica**. Dissertação de Mestrado. PUC-Rio em 2004.

MOURA, S. S.; SCHWABE, D.; **Interface Development for Hypermedia Applications in the Semantic Web**, Joint Conference 10th Brazilian Symposium on Multimedia and the Web & 2nd Latin American Web Congress, Ribeirao Preto, SP, Brazil pp 106-113, IEEE Computer Society, October 2004, ISBN: 0-7695-2237-8.

NUNES, D. A. **HyperDE - um Framework e Ambiente de Desenvolvimento dirigido por Ontologias para Aplicações Hipermídia.** Dissertação de Mestrado, PUC-Rio em 2005.

PANDEY, K. **Business Rules in User Interfaces.** Business Rules Journal, Vol. 8, No. 12, Dec. 2007. Disponível em <http://people.ischool.berkeley.edu/~glushko/ IS243Readings/BusinessRulesInUIs.pdf>. Acesso em 14/01/2013.

PATERNO F.; SANTORO C.; SPANO L. D. MARIA: A Universal, Declarative, Multiple Abstraction Level Language for Service-Oriented Applications in Ubiquitous Environments, ACM Transactions on Computer-Human Interaction (TOCHI), Volume 16 Issue 4, November 2009, Article No. 19, 2009

PAULHEIM, H., **An Ontology of User Interfaces and Interactions**, Ontology-based Application Integration, pp 119-150, 2011

POHJA, M., **Comparison of common XML-based Web user interface languages**, Journal of Web Engineering, volume 9, number 2, pages 95-115, 2011.

REZENDE, S. O. **Sistemas Inteligentes - Fundamentos e Aplicações**, Editora Manole, primeira edição, 2003

ROTH, F. H. **Rule-Based Systems**. Communications of the ACM, Volume 28, Number 9, September 1985.

SANTOS, D. L. **Um Modelo de Operações para aplicações na web semântica**. Dissertação de Mestrado. PUC-Rio, 2010.

SOKOLOV, V. **A modular approach to an ontology-based web application framework**. Master's Thesis. Tallinn University of Technology, Faculty of Information Technology, Department of Computer Science, 2009

SOKOLOV, V. **Wongi-Engine - A rule engine written in Ruby**. 2012. Disponível em < https://github.com/ulfurinn/wongi-engine>. Acesso em 11/10/2012.

THEVENIN, D., COUTAZ, J. **Plasticity of User Interfaces: Framework and Research Agenda**. Human–Computer Interaction — INTERACT'99, Published by IOS Press, IFIP TC.13, 1999.

VANDERDONCKT, J. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. Proc. of 17th Conf. on Advanced Information Systems. Vol. 3520. Springer-Verlag, Berlin: O. Pastor & J. Falcão e Cunha (eds.), Lecture Notes in Computer Science, 2005.

Apêndice I – Lista de Widgets Concretos e extensões

Widgets Concretos disponíveis até o momento

DHTMLXSchedulerAnchorAjaxDialog

DHTMLXSchedulerComposition

DHTMLXSchedulerEndDate

DHTMLXSchedulerEntry

DHTMLXSchedulerLineHead

DHTMLXSchedulerStartDate

DHTMLXSchedulerText

HTMLAnchor

HTMLComposition

HTMLForm

HTMLFormButton

HTMLFormCheck

HTMLFormInput

HTMLFormRadio

HTMLFormSelect

HTMLFormText

HTMLHeading

HTMLHorizontalLine

HTMLImage

HTMLLabel

HTMLLineBreak

HTMLList

HTMLListComposition

HTMLListItem

HTMLPage

HTMLParagraph

HTMLSpan

HTMLText

JqueryAjaxAnchor

JQueryAjaxAnchorDialog

JQueryAjaxAutocomplete

JQueryAjaxChainedSelect

JQueryAjaxGetRemote

JQueryAnythingSlider

JqueryDatePickerInput

JQueryIncrementerFormInput

JQueryTempoTemplateEngine

SHDMNavigationAnchor

Extensões Concretas disponíveis até o momento

JQueryCopyDateTo

JQueryCopyTo

JQueryFormAjax

Apêndice II - Exemplo de esquema de interface com busca no twitter

```
interface schema = {
      :name => 'main_page',
      :node_content => {:concrete_widget => "HTMLPage", :params => {:title => "My
Demo page", :include_css=> ["css/default_interface.css"] }},
      :children => [
        { :name => 'header',
          :node_content => {:concrete_widget => "HTMLComposition", :params =>
{:css_class => "header"}},
          :children => [
            { :name => "main_heading",
              :node_content => { :concrete_widget => "HTMLHeading", :params =>
{:number => 1,
              :content => "Twitts about Anything", :css_class => "heading2"}}
            }
          ]
        },
        {
          :name => 'form_search',
          :node_content => {
            :concrete_widget => "HTMLForm",

            :params => {:css_class => "box center", :action =>
"http://search.twitter.com/search.json?&callback=?", :method => "get"}
          },
          :children => [
            {
              :name => 'label_search',
              :node content => {:concrete widget => "HTMLLabel",
              :params => {:content => "Search: "}},

            },
            {
              :name => 'query',
              :node_content => {:concrete_widget => "HTMLFormInput",
              :params => {:name => 'q', :css class => "input", :content => "type
query"}},

            },
            {
              :name => 'items_per_page_label',
              :node content => {:concrete widget => "HTMLLabel",
              :params => {:content => "Items per page: "}},

            },
            {
              :name => 'items per page',
              :node content => {:concrete widget => "HTMLFormSelect",
              :params => {:name => 'rpp', :css_class => "input", :options => ["5",
"10", "15", "20", "25", "30"] }},

            },
```

```
            {
              :name => 'btn_send',
              :node_content => {:concrete_widget => "HTMLFormButton",
              :params => {:css_class => "input", :content => "Go"}},

            }
          ]
        },
        {
          :name => 'tweets',
          :node_content => {
            :concrete_widget => "JQueryTempoTemplateEngine",
            :params => {
              :url_from_element_id => 'url_search_twitter',
              :node_json_result_element => "['results']", :msg_error => "Sorry"
            }
          },
          :children => [
            {
              :name => 'twitt',
              :node_content => {:concrete_widget => "HTMLListItem", :params =>
{:css_class => "row"}},
              :children => [
                { :name => "columnA",
                  :node_content => { :concrete_widget => "HTMLComposition",
:params => {:css_class => "column grid_4 user"} },
                  :children => [
                    { :name => "profile image url",
                      :node_content => { :concrete_widget => "HTMLImage", :params
=> {:content => "{{profile_image_url}}"}}
                    },
                    { :name => "from user",
                      :node_content => { :concrete_widget => "HTMLHeading",
:params => { :number => 3, :content => "{{from user}}"}}
                    },
                ]},
                { :name => "columnB",
                  :node content => { :concrete widget => "HTMLParagraph", :params
=> {:content => "{{text}}", :css_class => "column grid_8"} },
                  :children => [
                    { :name => "created at",
                      :node content => { :concrete widget => "HTMLSpan", :params
=> {:content => ",{{created_at | date '\\at HH:mm on EEEE' }}", :css_class =>
"time"}}
                    }
                  ]
                },

              ]
            }

          ]
        },
        { :name => 'footer',
          :node_content => {:concrete_widget => "HTMLComposition", :params => {}},
          :children => [
            { :name => "footer text",
              :node_content => { :concrete_widget => "HTMLParagraph", :params =>
{:css_class => "row",
              :content => "http://twigkit.github.com/tempo"}}
            }
          ]
```

```
        },
      ]
    }

 extensions= [
      {:name => 'ext2', :extension => 'JQueryFormAjax', :nodes => ['form_search'],
:params => {:target => "tweets"}},
    ]
```

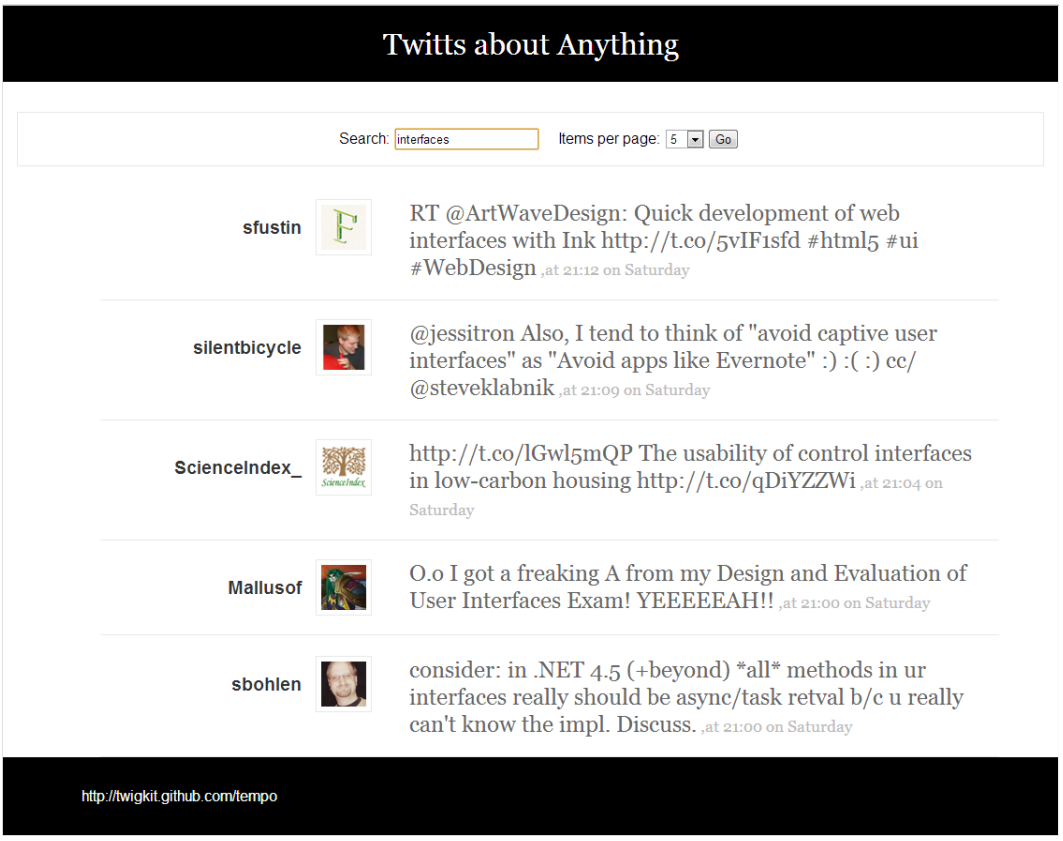

Figura 48 – Interface resultante gerada pelo intepretador

Apêndice III - Comparativo entre as UIDLs

Características gerais das UIDLs

UIL	Specificity	Publicly available	Type	Weight of the organization behind	Level of usage
DISL	Multimodal UIs for mobile devices	2	Research	Paderborn University	3
GIML	Multimodal	3	Research	Technical University of Dresden and Leipzig University of Applied Sciences	2
ISML	GUI, multiplatform, multidevice	2	Research	Bournemouth University	1
RIML	Mobile devices	0	Industry	Industry: SAP Research, IBM Germany, and Nokia Research Center along with CURE, UbiCall, and Fuijitsu Invia	3
SeescoaXML	Multiplatform, multidevice, dynamic generation UI	2	Research	Expertise Centre for Digital Media Limburgs Universitair Centrum	3
SunML	Multiplatform	4	Research	Rainbow team, Nice University	3
TeresaXML	Multiplatform, multidevice,	4	Research	HCI Group of ISTI-C.N.R.	3
UIML	Multiplatform	4	Industry	Harmonia, Virginia Tech Corporate Research (OASIS)	3
UsiXML	Multiplatform	5	Research	UCL	3
WSXL	multiplatform, multidevice	4	Industry	IBM	3
XICL	Multiplatform	3	Research	Federal University of Rio Grande do Norte, Brazil	3
XIML	multiplatform, multidevice	4	Research	Redwhale Software	3

Garcia et al. [2009]

Comparação das propriedades das UIDLs

UIL	Models	Methodology	Tools	Supported languages	Supported platforms	Level	Tags	Concepts
DISL	Presentation, dialog and control	Specification of a generic, platform-independent multimodal UI	Rendering engine	VoiceXML, Java MIDP, Java Swing, Visual C++	Mobile and limited devices	Model level	Not specified	Head element, interface classes (structure, style, behavior), state, generic widgets
GIML	Presentation, dialog, and domain	Specification of a generic interface description.	GITK (Generalized Interface Toolkit)	C++, Java, Perl	Not specified	Meta-model	15 tags	Interface, dialog, widget, objects
ISML	Presentation, task, dialog, domain	Specification of a generic UI description	Under construction	Java, Microsoft foundation class, Java swing classes	Desktop PC, 3D screen	Model level	Not specified	Mappings and constrains, action events, meta-objects, display parts, controller parts, interaction definition
RIML	There is no information	Specification of a generic UI description	There is no information	XHTML, XFORMS, XEvents, WML	Smart phone, pda, Mobile, Desktop Pc	Model level	There is no information	Dialog, Adaptation, layout, element
Seesco aXML	Task, Presentation, dialog	Specification of a generic UI description	CCOM (BetaVersion 1.0 2002) PacoSuite MSC Editor	Java AWT, Swing, HTML, java.microedition, applet, VoxML, WML Juggler	Mobile, desktop PC, Palm III	Model level	Not specified	Component, port, connector, contract, participant, blueprint, instance, scenario, pltform, user, device
SunML	Presentation, dialog, domain	Specification of a generic UI description	SunML Compiler	Java Swing, voiceXML, HTML, UIML,	Desktop Pc, pda	Model level	14 tags	Element, list, link, dialog, interface, generic events, synchronization
Teresa XML	Presentation, task, dialog	Specification of a generic UI description	CTTE Tool for task Models Teresa	Markup: Digital TV, VoiceXML, XHTML/SVG, X+V Programming: C#	DigitalTV, Mobile, Desktop PC,	Model level	19 tags	Mappings, models, , platform, task, input, output
UIML	Presentation, dialog, domain	Specification of a generic UI description	UIML.net, VoiceXML renderer, WML renderer, VB2UMIL	HTML, Java, C++, VoiceXML, QT, CORBA, and WML	desktop PC, a handheld device, tv, mobile	Model level	50 tags	interconnection of the user interface to business logic, services
WSXL	Presentation,dialog , domain	Specification of a generic UI description	Not specified	HTML	PC, Mobile phone,	Model level	12 tags	CUI=XForms, WSDL, Mapping=XLang Workflow=WSFL, Logic=XML event
XICL	Presentation,dialog ,	Specification of a generic UI description	XICL STUDIO	HTML, ECMAScript, CSS e DOM.	desktop PC	Model level	Not specified	Component, structure, script, events, properties, interface
XIML	Presentation, task, dialog, domain	Specification of a generic UI description	XIML Schema	HTML, java swing, WLM	Mobile, desktop PC, PDA	Model level	32 tags	Mappings, models, sub models, elements, attributes and relations between the elements

Garcia et al. [2009]