UNIVERSIDADE FEDERAL DA PARAÍBA

ANDREA FERNANDA FONTES BEZERRA

Geração de *layout* de interfaces gráficas baseado em ontologias para documentos do Registro Eletrônico em Saúde

João Pessoa

2014

ANDREA FERNANDA FONTES BEZERRA

Geração de *layout* de interfaces gráficas baseado em ontologias para documentos do Registro Eletrônico em Saúde

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal da Paraíba como requisito parcial para obtenção do título de Mestre em Informática.

Área de concentração:
Computação Distribuída

Orientador:
Prof. Dr. Gustavo Henrique Matos Bezerra Motta

João Pessoa

2014

B574g Bezerra, Andrea Fernanda Fontes.
Geração de layout de interfaces gráficas baseado em ontologias para documentos do Registro Eletrônico em Saúde / Andrea Fernanda Fontes Bezerra.-- João Pessoa, 2014.
111f.
Orientador: Gustavo Henrique Matos Bezerra Motta
Dissertação (Mestrado) – UFPB/CI
1. Informática. 2. Geração automática – interfaces gráficas – usuários. 3. Layout – interfaces gráficas – usuários.
4. Registro eletrônico de saúde – documentos – ontologia.

UFPB/BC CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de **ANDREA FERNANDA FONTES BEZERRA**, candidato ao Título de Mestre em Informática na Área de Sistemas de Computação, realizada em 23 de maio de 2014.

Ao vigésimo terceiro dia do mês de maio do ano dois mil e quatorze, às quatorze horas, na Escola Superior de Redes - Universidade Federal da Paraíba - reuniram-se os membros da Banca Examinadora constituída para examinar a candidata ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Computação Distribuída*", a Sra. **ANDREA FERNANDA FONTES BEZERRA**. A comissão examinadora foi composta pelos professores doutores: GUSTAVO HENRIQUE MATOS BEZERRA MOTTA (PPGI-UFPB), Orientador, CLAUIRTON DE ALBUQUERQUE SIEBRA (PPGI-UFPB), examinador interno e Presidente da Banca, e MARCO ANTÔNIO GUTIERREZ (USP), como examinador externo. Dando início aos trabalhos, o professor CLAUIRTON DE ALBUQUERQUE SIEBRA cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou à palavra a candidata para que a mesma fizesse, oralmente, a exposição do trabalho de dissertação intitulado *"Geração de layout de interfaces gráficas baseado em ontologias para documentos do Registro Eletrônico em Saúde"*. Concluída a exposição, a candidata foi argüida pela Banca Examinadora que emitiu o seguinte parecer: "A*provado*". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, eu, Alisson Vasconcelos de Brito, Coordenador do PPGI, servindo de secretário, lavrei a presente ata que vai assinada por mim e pelos membros da Banca Examinadora. João Pessoa, 23 de maio de 2014.

Alisson Vasconcelos de Brito

Prof. Dr. Gustavo Henrique Matos Bezerra Motta
Orientador (PPGI-UFPB)

Prof. Dr. Clauirton de Albuquerque Siebra
Examinador Interno (PPGI-UFPB)

Prof. Dr. Marco Antônio Gutierrez
Examinador Externo (USP)

AGRADECIMENTOS

Em primeiro lugar a Deus, por ter me dado forças para persistir e por ter me guiado da melhor forma possível na busca pelos meus objetivos.

À minha mãe (*in memoriam*), por todo o apoio e incentivo, mesmo durante o pior período de sua vida, no qual ainda reunia forças para me transmitir paz e esperança.

Ao professor Gustavo Motta, pela sua orientação presente, confiança no meu trabalho e extrema compreensão em todos os momentos.

Aos meus companheiros do LArqSS: Luciano, Rodrigo, Hugo, Duílio, Pizzol e João, sem os quais eu não conseguiria alcançar os objetivos pretendidos no trabalho.

À FINEP, pelo fomento ao projeto OpenCTI.

RESUMO

BEZERRA, A. F. F. **Geração de *layout* de interfaces gráficas baseado em ontologias para documentos do Registro Eletrônico em Saúde**. 111 f. Dissertação (Mestrado) – Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2014.

A informática em saúde apresenta muitos desafios a serem superados. Um de seus principais ramos de pesquisa são os Registros Eletrônicos em Saúde (RES), responsáveis, dentre outros, pelo armazenamento, exibição e manipulação de registros clínicos do paciente. Sistemas deste tipo requerem flexibilidade do domínio da aplicação, de modo que alterações nos documentos do RES sejam realizadas em tempo de execução, sem recompilação ou reimplantação da aplicação, por exemplo, em um servidor *web*. Abordagens da literatura propõem modelos genéricos de representação de domínio e apresentação, sem definições ontológicas de *layout* e estilo de interface com o usuário (UI). Estes, quando bem organizados, melhoram a aceitação do sistema pelos usuários. Este trabalho teve como objetivo o desenvolvimento de um *framework* para geração de *layout* e estilo de interface gráfica com o usuário para documentos do RES, baseado em ontologias Web Ontology Language (OWL), com uso de *restrições*. Através da centralização e combinação dos metadados biomédicos e de documentos para o RES, foi possível aplicar *layout* e estilo para os documentos do RES, com uso de *grids*, com definição ontológica adicional de formatos de apresentação para a área médica, facilitando o desenvolvimento da UI para o RES a manutenção da interface gráfica da aplicação.

Palavras-chave: geração automática de interfaces gráficas com o usuário, *layout* de interfaces gráficas com o usuário, registro eletrônico de saúde, ontologia.

ABSTRACT

BEZERRA, A. F. F. **Ontology-based graphical user interface layout generation for the electronic health record**. 111 f. Dissertation (Masters) – Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2014.

Health informatics is a domain that presents several challenges to be overcome. Electronic Health Records (EHR) are one of its most important subdomain, in charge of storage, exhibition, and manipulation of patient clinical information, among others. EHR systems require domain flexibility, which allows modifications in the structure of documents without application recompilation or redeployment, for instance, in a web server. Current approaches in the literature propose generic models to represent domain and presentation, without ontological definitions for user interface (UI) layout and style. These, when properly organized, improve the acceptance of the system by users. This work aims to develop a framework to layout and style generation for graphical user interface of EHR documents, based on Web Ontology Language (OWL) ontologies and using restrictions. By centralizing and combining metadata from biomedical and documents domains, it was possible to apply layout and style to EHR documents, with the use of grids, including additional ontological definition of presentation formats for the medical field, facilitating UI development and maintenance.

Keywords: automatic graphical user interface generation, graphical user interface layout, electronic health record, ontology.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AAC	Atribuição automática de componentes (refere-se ao algoritmo)
AJAX	*Asynchronous Javascript XML Request*
API	*Application Programming Interface*
ARIA	*Accessible Rich Internet Application*
CBU	*Content Binding Unit*
CSS	*Cascading Style Sheet*
DOM	*Document Object Model*
DCU	*Display Content Unit*
EHR	*Electronic Health Record*
GoF	*Gang of Four*
GUI	*Graphical User Interface*
HTML	*Hypertext Markup Language*
HULW	Hospital Universitário Lauro Wanderley
JSF	*Java Server Faces*
JSON	*JavaScript Object Notation*
LGPL	*Lesser General Public License*
LTR	*Left to Right*
MVC	*Model-View-Controller*

OO	*Orientação a Objetos*
OOUI	*Object Oriented User Interface*
OWL	*Web Ontology Language*
PEP	Prontuário Eletrônico do Paciente
PNG	*Portable Network Graphics*
RES	Registro Eletrônico do Paciente
RIA	*Rich Internet Application*
RTL	*Right to Left*
UI	*User Interface*
UIML	*User Interface Markup Language*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*
WIMP	*Windows Icons Menus Pointers*
XML	*Extensible Markup Language*

SUMÁRIO

1

Introdução

O termo prontuário tem sua etimologia no latim *promptuarium*, que “significa lugar onde se guardam ou depositam as coisas que se pode precisar ou devem estar disponíveis a qualquer instante” (MASSAD et al., 2003, p. 43). Na medicina, o prontuário do paciente é essencial na prestação de serviços de saúde dos indivíduos, reunindo de forma sistemática as informações imprescindíveis para a continuidade e a qualidade dos tratamentos prestados ao paciente.

As expressões prontuário nosológico do paciente ou prontuário médico do paciente são mais expressivas, completas e formais, podendo ser empregadas em trabalhos científicos (BACELAR, 2012). Entretanto, a denominação *prontuário do paciente*, mais simplificada, é perfeitamente aceitável na rotina de trabalho e reflete melhor o contexto jurídico, de acordo com a Constituição Federal do Brasil (BRASIL, 1988) e do Código de Ética Médica (CONSELHO FEDERAL DE MEDICINA, 2009), que determinam que as informações do prontuário pertencem única e exclusivamente ao paciente, devendo ser mantidas em sigilo, garantindo-se, assim, a preservação do direito à intimidade do mesmo.

A verificação da evolução da saúde do paciente se dá através da análise das informações anotadas no seu prontuário, tais como diagnósticos realizados, procedimentos aplicados, medicamentos prescritos, melhoria ou não na condição clínica do indivíduo, condutas terapêuticas associadas, dentre outras (MASSAD et al., 2003).

O prontuário em papel foi adotado há muitos anos e gradativamente vem sendo implementada a sua versão eletrônica, chamada de prontuário eletrônico do

paciente (PEP), na expectativa de melhorar a qualidade dos registros médicos, de possibilitar o acesso rápido à informação e de eliminar a necessidade de espaços físicos para a manutenção de prontuários em papel.

A evolução das tecnologias de informação e comunicação permitiu o compartilhamento de informações. Com isso, o PEP, de uso exclusivo e interno da instituição de saúde, passou a se tornar um Registro Eletrônico de Saúde (RES), compartilhável entre instituições dentro de uma região (município, estado ou país) (CONSELHO FEDERAL DE MEDICINA, 2012).

O prontuário eletrônico, bem como o registro eletrônico em saúde, é um processo, não um produto ou um *software*. Demanda tempo para sua adoção, pois necessita de adaptação de processos existentes de e pessoas dentro de uma instituição de saúde. Requer novos investimentos em acompanhamento e gestão da produção, custo e qualidade, sendo essencial a padronização da estrutura e da terminologia da área médica, além de uma definição de protocolos de comunicação entre instituições, de métodos armazenamento de dados e, ainda, de segurança da informação (MASSAD et al., 2003).

1.1 Motivação

A informática em saúde é uma área inovadora e que apresenta grandes problemas de implementação a serem superados. Este trabalho é motivado por algumas características essenciais aos sistemas de RES que precisam ser aprimoradas e que ainda não possuem um padrão *de facto* adotado pela indústria. São elas: usabilidade, flexibilidade requerida pelo domínio da aplicação, contexto, interoperabilidade.

1.1.1 Usabilidade

Estudos recentes mostram que o uso de sistemas de RES para uma série de atividades clínicas que, em princípio, deveriam estar facilitadas ainda é muito baixo (MASSAD et al., 2003). A adoção tem sido lenta e um dos motivos principais alegado por médicos é que o uso de sistemas RES requer mais tempo para o

preenchimento das informações (LO et al., 2007). Soma-se a esses fatores a baixa usabilidade dos sistemas de RES relatada pelos profissionais de saúde como sendo um dos maiores obstáculos para a sua ampla adoção. Ainda hoje, grande parte dos profissionais da área prefere trabalhar com documentos em papel, devido à dificuldade de adaptação ao novo fluxo de trabalho imposto pela implantação dos sistemas informatizados (DUARTE, 2011).

É importante ressaltar que os sistemas de RES coletam muitas informações de diversas categorias sobre a saúde do paciente. Neste caso, muitos campos de entrada de dados na tela podem causar fadiga ao usuário final desses sistemas, o que requer que os mesmos possuam interfaces gráficas (*Graphical User Interface* - GUI) muito bem projetadas (INSTITUTE OF MEDICINE, 2003). Isto pode ser conseguido com uma abordagem de design centrado no usuário, cujo foco não é somente na usabilidade das interfaces gráficas, mas também em sua utilidade para a realização das tarefas.

Considerando o ponto de vista dos desenvolvedores, em muitos projetos de *software*, grande parte do tempo de desenvolvimento é dedicada à construção de interfaces gráficas (KENNARD; LEANEY, 2010). Um *layout* eficaz é um dos aspectos mais importantes da apresentação da informação. A ampla maioria dos *layouts* criados para GUI é definida inteiramente à mão, onde um *designer* toma as decisões relacionadas ao posicionamento, dimensões e adornos dos objetos de interface. Em geral, é um processo custoso e inadequado para situações onde informações urgentes devem ser comunicadas (LOK; FEINER, 2001).

Portanto, a automatização deste processo é capaz de trazer nítidos benefícios. Contudo, é uma tarefa difícil principalmente devido à diversidade de arquiteturas, plataformas e ambientes de desenvolvimento das aplicações, o que impede ampla adoção ou padronização pela indústria de *software* (KENNARD; LEANEY, 2010).

1.1.2 Flexibilidade requerida pelo domínio da aplicação

Aplicações para o registro eletrônico em saúde, diferentemente da maioria dos sistemas de informática, possuem características peculiares, pois o domínio da

aplicação está sujeito a constantes mudanças devido a avanços científicos e novas descobertas na área médica (DUARTE, 2011). Para a coleta de informação no âmbito de um hospital, por exemplo, formulários de papel constituem um *framework* natural para criação de interfaces que são familiares aos profissionais de saúde.

Formulários eletrônicos, por sua vez, apresentam ao usuário a possibilidade de inserir dados em caixas de texto ou listas de seleção, por exemplo, através de um teclado ou dispositivo apontador (mouse, canetas etc). Tais formulários são capazes de analisar os dados inseridos, interagindo com os usuários e sugerindo correções sintáticas ou até mesmo semânticas, no caso de sistemas de suporte à decisão clínica (CDS, *Clinical Decision Support*) (GREENES, 2006).

Consideremos, então, um cenário específico: num hospital existe um formulário de evolução médica com 30 campos disponíveis para o preenchimento. Com surgimento de novos medicamentos ou a ocorrência de uma epidemia, por exemplo, pode ser necessário que sejam coletados certos dados adicionais com maior frequência do que antes dessas circunstâncias. Os formulários de papel já impressos muito provavelmente não serão descartados, mas o profissional terá que se lembrar de todos os novos dados necessários para coleta, escrevendo-os como texto livre em um campo para observações. Neste caso, a inserção de novos campos em uma ficha de evolução médica trará a necessidade de que o sistema reflita tais mudanças e que, de maneira compreensível, possa exibir fichas antigas e fichas novas (com novos campos) de modo compreensivo ao usuário (VAN DER LINDEN et al., 2009).

Portanto, diante do exemplo mencionado, percebe-se que sistemas de RES requerem alterações constantes na interface com o usuário (UI, *User Interface*) para refletir as alterações imprevisíveis do domínio. Interfaces gráficas manualmente codificadas requerem a presença ou o trabalho do *designer* e ainda que seja feita a recompilação e/ou reimplantação da aplicação no caso de servidores *web*. Um sistema que permita as alterações mencionadas na interface gráfica em tempo de execução trará nítidos benefícios aos desenvolvedores, ao trabalho dos profissionais de saúde e ao paciente.

1.1.3 Contexto

Com a evolução e ampla adoção da computação ubíqua, os sistemas informatizados para o RES estão sendo adaptados a essa realidade. A popularização de dispositivos como telefones celulares, *smartphones*, *netbooks, tablets*, etc, tem despertado o interesse da comunidade médica para o uso dessas tecnologias no atendimento ao paciente. O prontuário eletrônico pode estar acessível a partir de tais dispositivos, que, entretanto, possuem características especiais, como menores resoluções de tela, que requer *layout* diferenciado de interfaces gráficas, menor poder computacional e estão também sujeitos a falhas de segurança. Tais dispositivos ainda podem ter acoplados a si diversos tipos de sensores, câmeras fotográficas, microfones, o que os torna particularmente úteis para uso em medicina.

Entretanto, os sistemas de RES devem estar preparados para lidar com o contexto do usuário. Para isto, os sistemas de RES devem:

- Identificar o equipamento com o qual o usuário está acessando o sistema.
- Tomar a decisão para exibir interfaces gráficas com componentes diferenciados, adequados às baixas resoluções de tela, omitir informações ou truncar termos clínicos quando apropriado para não sobrecarregar a tela, compactar imagens quando pertinente, etc.
- O sistema deve também usar dados obtidos de sensores e outros dispositivos para determinar o nível de segurança do usuário naquele momento, permitindo ou não que o mesmo realize determinadas operações sobre o prontuário do paciente (segurança sensível ao contexto). Informações avançadas de contexto podem ser úteis para determinação do *layout* da UI, como a distância do usuário da tela e *eyetracking* (LOK; FEINER, 2001).

1.2 Objetivos

O presente trabalho tem o objetivo geral de estender o *framework* já existente para geração de UI para sistemas de RES, o MedViewGen (DUARTE, 2011), especificamente para melhorar a usabilidade da UI ao incluir funcionalidades que faltam ao MedViewGen, como a geração de *layout* para os documentos, não

somente com relação ao posicionamento de componentes, mas também suas dimensões, e estilo.

Como objetivos específicos, temos:

Objetivo 1. Criar estruturas de *layout* pré-definidas baseadas em convenções médicas para formatação de arquétipos específicos de documentos do RES, para tornar a apresentação da informação padronizada.

Objetivo 2. Reestruturar o modelo ontológico de documentos e outros modelos já existentes do *MedViewGen* (DUARTE, 2011), para permitir a geração de *layout* e estilo para documentos do RES, com representação através da linguagem de ontologias OWL (Web Ontology Language) (W3C WORLD WIDE WEB CONSORTIUM, 2012).

Objetivo 3. Estender a API do *MedViewGen* (DUARTE, 2011) para permitir o alcance dos objetivos anteriores.

Objetivo 4. Integrar a solução desenvolvida a um sistema RES, de modo a demonstrar a aplicabilidade dos métodos e da arquitetura utilizados na ferramenta. Utilizaremos para isso o sistema OpenCTI[1].

1.3 Escopo e limitações do gerador de *layout*

Mesmo que um sistema realize a geração de UI, não é possível excluir a função do *designer* de interface, especialmente durante as fases de projeto*,* mas sim diminuir a necessidade imediata deste em tempo de execução, em virtude da natureza de certas aplicações que não podem esperar sua disponibilidade nem ser recompiladas com frequência.

Os modelos ontológicos para apresentação da informação/UI têm como escopo documentos individuais (formulários), não abrangendo toda a interface da

[1] OpenCTI: Software de uma Central de Telemedicina para Apoio à Decisão Médica em Medicina Intensiva. Projeto financiado pela FINEP nº 01.08.0533.00.

aplicação. Itens de navegação, posição de menus e aparência e usabilidade da aplicação em geral são muito específicos de cada tipo de dispositivo (KENNARD; LEANEY, 2010) e, portanto, foram mantidos fora do objetivo deste trabalho.

Outro ponto importante ao se projetar UI para o RES deve levar em consideração os seguintes fatores:

- A resistência dos usuários ao uso de novas tecnologias;
- A usabilidade e a estética da UI;
- A consistência da UI;
- A segurança das informações do paciente;

Pode-se considerar que as duas últimas estão relacionadas, pois interfaces gráficas inconsistentes podem causar erros de preenchimento, levando ao armazenamento de informações incorretas. Neste caso, documentos para o RES têm requisitos de consistência da apresentação e segurança da informação do paciente que não permitem que o usuário ou mesmo o sistema, neste caso de modo automático, tenha liberdades para personalizar ou remodelar a UI a ponto de torná-la diferenciada de maneira que altere a rotina de preenchimento durante um expediente de trabalho.

Portanto, é necessário que o *layout* não sofra variações aleatórias na exibição de um documento em um mesmo tipo de dispositivo, devendo ser, na prática, modelado de acordo com as convenções estabelecidas dentro da instituição ou pelos usuários finais do sistema. Há *restrições* necessárias que limitam a disposição espacial dos diversos componentes na UI de um documento clínico, como a ordem em que os campos aparecem na ontologia respectiva, o que restringe, em parte, o algoritmo de *layout.*

1.4 Justificativa

Layouts de UI organizados melhoram a usabilidade e a estética das interfaces gráficas dos documentos, contribuindo para uma maior aceitação do uso do sistema por parte dos profissionais de saúde, sendo esta a justificativa principal para o benefício imediato dos usuários da aplicação.

A inserção ou edição de *templates* de documentos com *layout* para o RES em tempo de execução, sem recompilação ou *reimplantação* da aplicação no servidor *web*, e sem necessidade imediata da função do *designer* de UI, torna a aplicação prática e, ainda, escalável.

Com a centralização dos metadados para UI e das *restrições* de *layout*/estilo em ontologias, facilita-se a manutenção do *software* e ainda permite reuso de código, seja de alguma linguagem de programação ou de marcação de interface.

A adoção de padrões da área de informática, especificamente ontologias OWL, facilitará a compreensão da modelagem semântica da ontologia de *layout*/estilo, tornando ainda a informação legível para máquinas.

Considerando que foi desenvolvido no Laboratório de Arquitetura e Sistemas de Software (LArqSS), Departamento de Informática, Universidade Federal da Paraíba (UFPB), o *framework MedViewGen* baseado em ontologias para geração automática de UI para sistemas de RES (DUARTE, 2011), em uso no projeto OpenCTI, optou-se por dar continuidade ao referido trabalho, abordando características não estudadas que se referem ao *layout*/estilo de componentes das interfaces gráficas, usando as API disponíveis, alterando-as de acordo com a necessidade.

1.5 Metodologia

Para o planejamento e execução do presente trabalho foram realizados diversos estudos, alterações de algumas ontologias e de implementações no sistema do OpenCTI, de modo que enumeramos os principais tópicos, conforme segue:

- Revisão da literatura através de busca de publicações nos principais engenhos da área computacional, que incluiu ScienceDirect, IEEE, SpringerLink e PubMed, no período entre 2007 e 2013, não se excluindo referências anteriores já encontradas em outras pesquisas;
- Estudo de ontologias;
- Reestruturação dos modelos ontológicos já desenvolvidos para conceitos biomédicos, apresentação da informação/componentes genéricos de UI e

para mapeamento de componentes de UI para tecnologia específica, de modo a incluir metadados e outros itens necessários para acrescentar as funcionalidades que faltam ao MedViewGen (*layout* e estilo de documentos);

- Estudo e extensão da API já implementada no OpenCTI, incluindo o MedViewGen (DUARTE, 2011);
- Testes e validação da técnica empregada.

1.6 Estrutura do trabalho

Esta dissertação está estruturada de acordo com a seguinte lista de capítulos:

- No Capítulo 2, “Fundamentação teórica”, são apresentados os conceitos fundamentais para o desenvolvimento do trabalho e uma revisão dos trabalhos relacionados.
- No Capítulo 3, “Modelos ontológicos”, são descritas as alterações nos modelos ontológicos realizadas para o alcance dos objetivos.
- No Capitulo 4, “Arquitetura e implementação”, é detalhada a arquitetura do gerador de interface gráfica, MedViewGen, desenvolvido por Duarte (DUARTE, 2011), observando suas limitações quanto ao *layout* e estilo para a UI gerada, bem como abordagem acrescentada no presente estudo para recuperação de metadados de *layout*/estilo e sua aplicação na UI.
- No Capítulo 5, “Resultados, discussão e comparação com outros trabalhos”, são comentados os resultados visuais obtidos e as demais alterações realizadas e, ainda, é apresentado um comparativo com outros trabalhos relacionados encontrados.
- No Capítulo 6, “Considerações finais”, são apresentadas conclusões e observações importantes, bem como possíveis trabalhos futuros a partir dos resultados obtidos e da experiência adquirida.

2

Fundamentação teórica

Este capítulo objetiva descrever conceitos fundamentais para o desenvolvimento do trabalho proposto, incluindo aqueles presentes na literatura relacionada. Na seção 2.1, definiremos o conceito e a estrutura lógica de documentos clínicos para o RES. Na seção 2.2 será fornecida uma explicação sobre ontologias. Na seção 2.3, serão comentados conceitos e observações importantes sobre interfaces gráficas, *layout* e técnicas para sua geração baseada em *restrições*. Na seção 2.3.5, foram resumidos os principais trabalhos encontrados sobre geração automática de UI/*layout*.

2.1 Documentos clínicos

Por documento entende-se a estrutura de informação, tanto para entrada quanto para saída de dados. Sistemas de documentos especificam quais dados devem ser incluídos nos documentos e como eles devem ser organizados. O propósito dos documentos é tanto a obtenção de dados do usuário quanto a sua apresentação ao mesmo.

Documentos podem ser impressos em papel e também representados digitalmente em uma tela de computador. A especificação da estrutura de um documento é uma forma de conhecimento. Padrões estão sendo desenvolvidos para tal, de modo a melhorar a qualidade das informações estruturadas, e o reuso quando apropriado. A falta de especificidade para o domínio médico e de padronização tem prejudicado o uso e compartilhamento de bases do conhecimento usadas em sistemas de documentação clínica (GREENES, 2006).

Considerando estruturas de documentos, o termo *Documentation Knowledge Element* (DKE) refere-se a uma unidade de um documento que não pode mais ser dividida, podendo ser referido como *Documentation Especification* (DE). *Knowledge Element Group* (KEG) é um grupo de DKEs, podendo ser aninhados, de modo que formem grupos de KEGs. Um KEG tem atributos relacionados à base lógica para a sua posição em um documento, seu propósito e outros aspectos que podem ser usados para indexá-lo e recuperá-lo num sistema de gerenciamento do conhecimento. A especificação de um KEG que rege sua aparência em um documento, em termos de *layout*, fonte, cor, tipos de respostas permitidas quando da entrada de dados, tipos de comportamento, é denominada *template* (GREENES, 2006).

Um arquétipo é um modelo formal e reusável de um conceito de domínio expressado na forma de *restrições* sobre dados, com a finalidade de permitir aos profissionais, no caso, de saúde, criarem definições para as estruturas de dados dos sistemas informatizados e, ainda, prover uma base inteligente para consulta de dados.

Um *template* de documento é um modelo de sua estrutura formado por um grupo de arquétipos possivelmente aninhados que especificam dados para serem coletados. Este modelo pode ter uma descrição através de ontologias, de modo que se torna legível por computadores. A seção seguinte descreve brevemente alguns conceitos importantes sobre ontologias, essenciais ao presente estudo.

2.2 Ontologias

Uma das promessas mais importantes da Web semântica é o seu potencial para interoperabilidade de dados. O compartilhamento e reuso de ontologias por provedores de informação e desenvolvedores de aplicações é crítico para a interoperabilidade desejada. Se aplicações e fontes de dados distintas usam o mesmo conjunto de termos bem definidos para descrição do domínio, será mais fácil a comunicação entre sistemas informatizados (D'AQUIN; NOY, 2012).

A OWL 2 é uma linguagem de marcação para ontologias da Web Semântica com significado formalmente descrito. Ontologias OWL 2 definem classes,

propriedades e indivíduos. São intercambiadas principalmente como documentos RDF (*Resource Description Format*) (W3C WORLD WIDE WEB CONSORTIUM, 2012).

Em OWL, na definição de uma classe, devem ser consideradas as características de interesse que os elementos a serem agrupados devem ter. As classes em sua forma mais simples se relacionam em forma de uma hierarquia taxonômica, ou seja, apenas de forma classificatória. Um fato importante sobre classes OWL é que todas descendem direta ou indiretamente da classe owl:*Thing*, nativa, que representa qualquer coisa.

Os indivíduos podem ser os elementos das classes, ou seja, utilizando-se a terminologia da orientação a objetos (OO), pode-se dizer que eles são as instâncias das classes.

Contudo, utilizando apenas classes e indivíduos, obtém-se uma semântica exclusivamente taxonômica. A OWL permite a descrição de propriedades, que especificam fatos sobre indivíduos por meio de assertivas sobre a classe ou os próprios indivíduos. Ainda, existem dois tipos de propriedades em OWL, as propriedades de dado (*datatype properties* ou *data properties*), que ligam dados em forma de valores de algum tipo a um indivíduo, e as propriedades de objeto (*object properties*), que relacionam dois indivíduos quaisquer descritos na ontologia. Usando a terminologia OO, pode-se dizer que *data properties* representam atributos e que *object properties* representam relacionamentos entre objetos (indivíduos).

Arquivos que descrevem as ontologias podem ser criados e editados com auxílio de softwares específicos, conforme descrito na seção 2.2.1.

2.2.1 *Software* para criação e edição de arquivos OWL

Os arquivos OWL possuem uma sintaxe própria definida pelo World Wide Web Consortium (W3C). A manipulação direta destes arquivos é sujeita a erros e inconsistências, portanto, faz-se necessário o uso de um software para criação e edição desses arquivos.

O *software Protégé* é um editor de arquivos de ontologias, gratuito e de código aberto, desenvolvido pela Escola de Medicina da Universidade de Stanford, Estados Unidos (STANFORD UNIVERSITY, 2014). Possui diversas funcionalidades, além da criação e edição de arquivos OWL 2, suporte a outros formatos de arquivos (RDF/XML, Turtle, OWL/XML), a exibição de grafos de indivíduos ontológicos instanciados, ferramentas colaborativas e ainda está disponível em versão *desktop* e *web*.

A Figura 1 exibe a captura da tela inicial do Protégé, com as funções básicas, como criar ou abrir arquivos de ontologias.

Figura 1 - Tela inicial do *software* Protégé

A Figura 2 mostra um exemplo de captura de tela do Protégé, para edição de arquivo da ontologia, aberto na aba *Entities*, onde são exibidos: a hierarquia de classes ontológicas (à esquerda), lista de indivíduos agrupados por tipo (mais abaixo, à esquerda) e lista de *data properties* (abaixo, à esquerda). No centro da tela, as anotações da classe selecionada, a descrição e lista de membros.

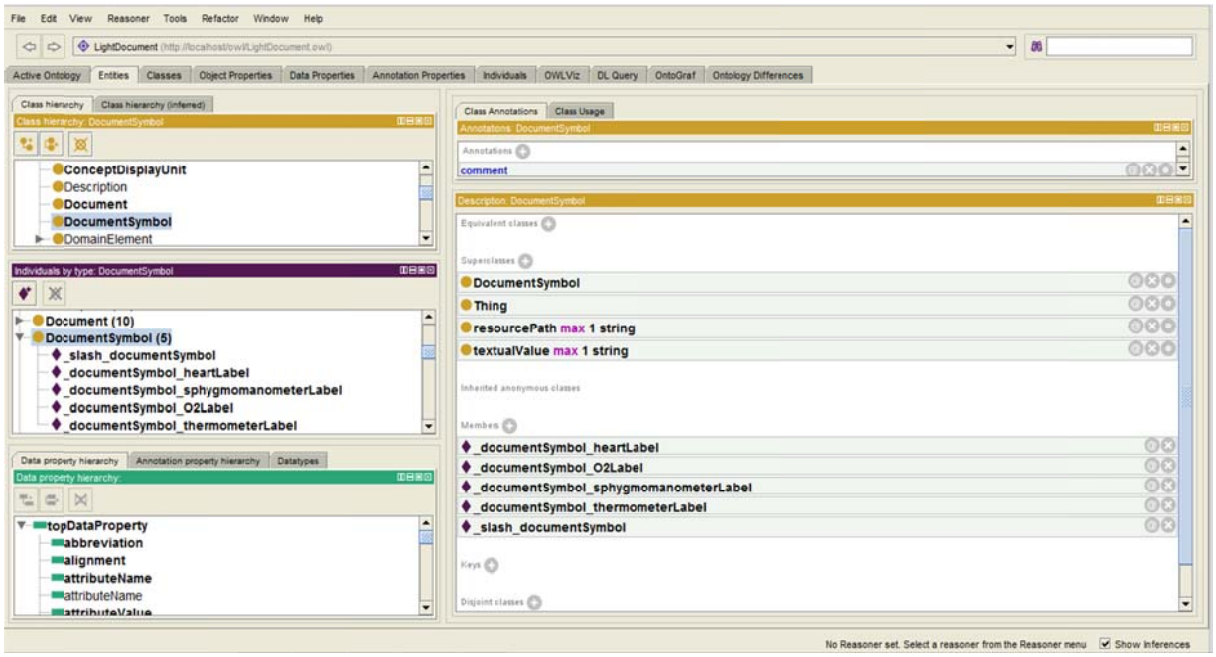

Figura 2 – Exemplo de captura de tela do Protégé

Modelos de componentes de interfaces gráficas também podem ser definidos ontologicamente. Na seção seguinte, explicaremos alguns conceitos importantes para o presente trabalho sobre UI, *layouts*, estilos e tecnologias *web* relacionadas.

2.3 Interface para o usuário

A interface para o usuário (UI, *user interface*) é uma parte de um sistema que permite a interação do usuário com a máquina, de modo que ele possa realizar suas tarefas (PREECE *et al,* 2002).

Ressalta-se que o termo *user interface* (UI) é mais abrangente e pode se referir a qualquer tipo de interface de um sistema, como por exemplo, interface de caracteres ou linha de comando. *Graphical User Interface* (GUI) refere-se

especificamente a interfaces gráficas compostas de elementos visuais, sendo que estas são mais complexas. Por simplicidade, usaremos daqui em diante neste trabalho o termo *user interface* (UI) como referência às interfaces gráficas estudadas.

Componentes de UI são os elementos genéricos que compõem uma interface gráfica, tais como acionadores de comandos (botões *etc*), itens para exibir informações ou permitir entrada de dados pelo usuário, como tabelas e caixas de entrada de texto, respectivamente. Tais componentes são conhecidos também como widgets (PREECE et al, 2002).

As interfaces gráficas podem ter sua estrutura genérica representada sob a forma de grafos do tipo árvore, onde os nós são componentes genéricos, definidos de acordo com a necessidade para execução de determinada tarefa pelo usuário ou exibição de dados ou outros itens. Refere-se aqui ao termo "*estrutura genérica*" porque tal representação não se relaciona com nenhuma tecnologia, mas apenas à estruturação lógica dos componentes no grafo que representa a interface gráfica. Metadados para *layout* e *formatação* de componentes podem ser eventualmente inseridos como nós no grafo. A árvore genérica de componentes pode ser convertida para uma árvore de componentes de uma tecnologia específica de desenvolvimento de interfaces gráficas. A Figura 3 exibe uma representação de uma estrutura genérica de UI para uma tela simples de *login*.

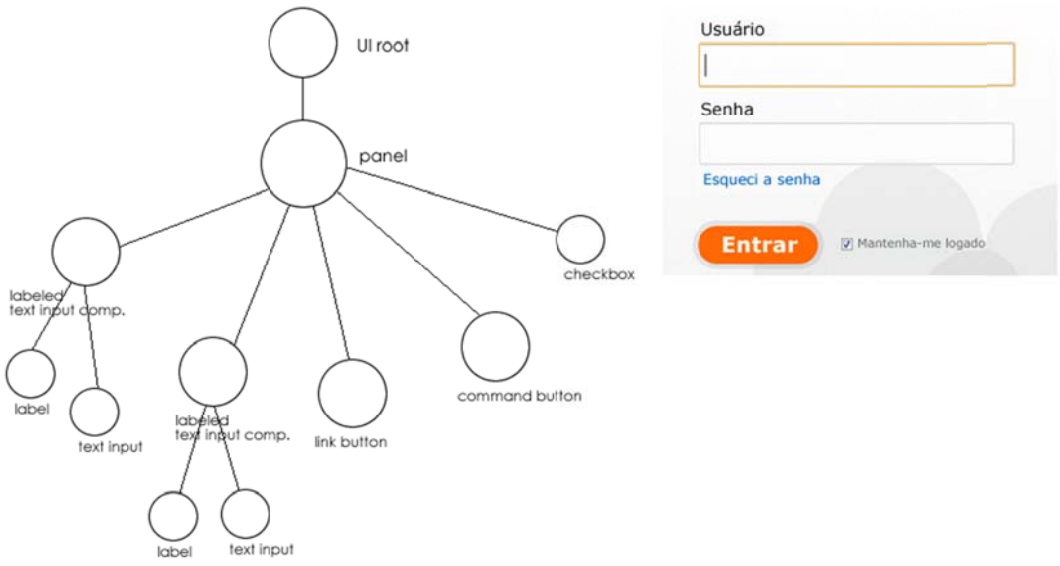

Figura 3 - Representação de UI genérica como árvore de componentes

No exemplo da figura anterior, o nó *UI root* é a raiz da interface gráfica, não necessariamente representada como componente visual, mas lógico. O componente genérico *panel* funciona como um agrupador e possui cinco componentes genéricos como filhos:

- *Labeled text input comp.* – componente genérico agrupador composto de dois filhos: rótulo (label) e campo de entrada de texto em única linha (text input) para inserção do nome de usuário.
- *Labeled text input comp.* - componente genérico agrupador composto de dois filhos: rótulo (label) e campo de entrada de texto em única linha (text input) para inserção da senha.
- *Link button* – componente genérico, neste caso, um botão com formato de *link* para abertura de nova janela com instruções em caso de esquecimento da senha pelo usuário.
- *Command button* – botão acionador de comando para fazer o *login*.
- *Checkbox* – caixa de seleção para marcar opção de permanecer com a sessão aberta por tempo maior.

Há diversas tecnologias para o desenvolvimento de interfaces gráficas. Em HTML (*HyperText Markup Language*) (W3C WORLD WIDE WEB CONSORTIUM, 1999), a linguagem de marcação padrão para interfaces web, uma página é composta de itens chamados *elementos*, representados por *tags*. O modelo convencional para interação com o documento foi padronizado pelo W3C como *Document Object Model* (DOM) (W3C WORLD WIDE WEB CONSORTIUM, 2003), que representa os elementos da página que contém a interface gráfica como uma árvore chamada *DOM tree.*

Para maior compreensão, os tópicos seguintes enumeram exemplos de componentes e seus correspondentes em algumas tecnologias específicas, a saber:

- Componente para entrada de texto com única linha – corresponde a uma caixa na horizontal que recebe o cursor para inserção de caracteres em uma única linha. Em HTML, corresponde ao elemento <input type=”text”> (W3C WORLD WIDE WEB CONSORTIUM, 1999). Em Java Swing, que é

uma tecnologia para plataformas *desktop*, o componente para entrada de texto corresponde ao objeto da classe JTextField.

- Componente para entrada de texto com múltiplas linhas – corresponde a uma caixa de texto que recebe o cursor para inserção de caracteres em múltiplas linhas. Em HTML, corresponde ao elemento <textarea>. Em Java Swing, corresponde ao objeto da classe JTextArea.
- Componente agrupador (*container*) – corresponde a um item que embora não seja necessariamente visível agrupa outros elementos em seu interior. Em HTML, corresponde a um elemento . Em Java Swing, ao objeto da classe JPanel.

Uma característica importante de UI é a sua usabilidade, que, em linhas gerais, se refere à facilidade com que o usuário consegue realizar uma tarefa específica e faz parte da área de estudo da Interação Homem-Computador (IHC). A UI pode motivar o usuário na utilização do sistema e, dependendo de suas características, tornar-se uma grande ferramenta para o usuário ou, se mal projetada, pode se transformar em um ponto decisivo na rejeição de um sistema (DUARTE, 2011).

Em fases finais do desenvolvimento de softwares, podem ser realizados testes com usuários reais para avaliar a usabilidade geral da UI. Um bom projeto de interfaces gráficas requer, dentre outras características, *layout* e estilo coerentes ao propósito da aplicação, discutidos nas seções seguintes.

2.3.1 *Layout*

Uma característica importante de uma UI é o *layout* dos seus componentes. *Layout* refere-se ao processo de determinação de posição e tamanho de componentes de UI exibidos em uma apresentação visual e o resultado do mesmo. O *layout* é um dos muitos itens de um projeto de apresentação da informação e deve complementar e/ou ser complementado por outras decisões que determinem a natureza dos objetos visuais apresentados. *Formato* significa como os objetos visuais são percebidos pelos usuários (como textos, gráficos etc) e seus atributos (cor, textura,

fonte etc) (LOK; FEINER, 2001). Geração automática de *layout* refere-se ao uso de componentes de *software* para automatização deste processo que parece ser a questão mais intratável na geração automática de UI (KENNARD; LEANEY, 2010).

É importante destacar que *layouts* de interfaces gráficas de computadores e outros dispositivos eletrônicos podem ter uma natureza dinâmica. A UI possui um ciclo de vida e, por isso, a estrutura de sua árvore virtual de componentes e seu *layout* podem ser alterados em tempo de execução, em consequência da interação do usuário e/ou como resultado de chamada a procedimentos ou funções. Por outro lado, *layouts* de documentos impressos ou páginas de revistas, por exemplo, possuem natureza estática, tendo sua estrutura definida somente uma vez, não havendo possibilidade de alteração.

Ressalta-se também a diferença entre estrutura da árvore genérica de componentes de UI e estrutura da UI concreta. Considerando tecnologias *web*, há *frameworks* Java, como JSF, cujos componentes são renderizados em HTML, sem que, entretanto, o desenvolvedor tenha controle da marcação utilizada (a menos que codifique seu próprio renderizador). Ainda que um documento possua poucos componentes de UI, a marcação HTML gerada pelo componente pode ser bem mais complexa, especialmente quando incorpora comportamento avançado. Por exemplo, um componente JSF do tipo *datatable,* para exibição de dados tabulares, poderia, teoricamente, ser equivalente a um elemento <table> em HTML. Contudo, como um *datatable* é um componente que requer funcionalidades avançadas em aplicações RIA (ações sobre as entradas, ordenação dos dados, etc), podem ser necessárias dezenas ou centenas de elementos <table> em HTML e outros elementos associados a diversas funções *Javascript* e regras de estilo para dar a apresentação final ao componente em páginas *web*.

Em resumo, uma *interface web* pode ter seu *layout* alterado sem que sua estrutura HTML o seja, assim como a sua estrutura de marcação HTML pode ser alterada e seu *layout* permanecer idêntico.

Há algumas técnicas reportadas na literatura para geração de *layouts*, sendo que as mais comuns são: *layout* baseado em *restrições* e aprendizagem de máquina (LOK; FEINER, 2001). Trataremos aqui somente a primeira por ser a mais adequada para a presente proposta.

2.3.1.1 *Layout* baseado em *restrições*

Um *layout* pode ser descrito através de um conjunto de *restrições*. Um gerenciador de *layout* determina a posição e o tamanho dos componentes de UI em tempo de execução, com base em uma política de *layout* que impõe *restrições*. Algumas políticas incluem arranjos em linhas ou colunas, espaçamento entre componentes, *layout* em *grids*, dentre outros.

Restrições abstratas são descrições de relacionamentos de alto nível entre componentes a serem inseridos na UI, tais como precedência de campos ou agrupamentos destes. Pode-se relacionar, ainda, *restrições* abstratas, onde certos componentes devem permanecer visualmente próximos para melhor caracterização e compreensão da informação apresentada, no caso deste trabalho, conceitos biomédicos em geral. Entretanto, tais *restrições* não determinam o tamanho nem a posição, inclusive no eixo z, de componentes na UI.

Restrições espaciais referem-se a valores ou configurações que forçam posicionamento ou dimensão de componentes da UI. São necessárias para melhorar a estética e a usabilidade. Por exemplo, um campo de entrada de dados não precisa ter uma dimensão maior que cinco ou seis caracteres para a inserção de um valor de pressão arterial, cuja faixa de valores possíveis - que não ultrapassa três caracteres - deve estar definida na ontologia biomédica respectiva, podendo ser usada para determinar as dimensões mínima e máxima do componente de UI. Outras formas de representar valores de pressão arterial (ou mesmo outros conceitos médicos) devem ser definidas ontologicamente, como em representações simplificadas amplamente aceitas (exemplo: 120 *mmHg (pressão arterial sistólica) e 80 mmHg (pressão arterial diastólica)* = 12/8) (VAN DER LINDEN et al., 2009). Trata-se, portanto, do que podemos chamar de *constraint* híbrida, tanto abstrata (porque estabelece uma relação lógica de precedência entre os dois valores) quanto espacial (porque altera posicionamento e dimensões dos componentes necessários para representar a informação) baseada no conhecimento biomédico, relacionada aos dados de entrada/saída. Neste caso, com *layouts* bem delimitados e/ou alternativos, pode-se organizar melhor o espaço disponível para acomodação de outros componentes, melhorando a aparência e usabilidade da interface.

Em sistemas de *grid*, cada página, tela ou contêiner de apresentação é dividido em um arranjo de células de dimensões definidas, que cada objeto de UI deve ocupar inteiramente. O problema com esta abordagem é que pode haver a necessidade de truncar um componente gráfico ou redimensioná-lo para se encaixar em seu espaço no *grid*, causando deformidades indesejadas.

Sistemas de geração de *layout* podem usar tanto *restrições* abstratas quanto espaciais ao mesmo tempo. Uma gramática com regras para *layout* pode ser definida ou ainda o sistema pode usar dados fornecidos pelo usuário. Em seguida, as *restrições* devem ser obtidas ou extraídas de diversas fontes. *Restrições* abstratas podem ainda ser extraídas a partir de relacionamento entre entidades de modelos de bancos de dados relacionais (LOK; FEINER, 2001), e, possivelmente, de modelos ontológicos e seus indivíduos.

2.3.2 Estilo

Para tecnologias *web*, o conceito de folhas de estilo surgiu com a primeira especificação denominada *Cascade StyleSheet* (CSS), publicada pelo W3C no ano de 1996 (W3C WORLD WIDE WEB CONSORTIUM, 1996). Uma folha de estilo consiste em um conjunto de propriedades pré-definidas do tipo chave=valor, definidas na própria especificação, com a finalidade de agrupar características de formatação associadas a elementos HTML ou a grupos destes.

As propriedades de estilo podem ser definidas dentro da página HTML através do elemento <style>, diretamente associadas ao elemento a ser formatado através dos atributos *style* e *class* e ainda em um arquivo separado, o que, neste último caso, melhora a legibilidade da marcação HTML e a manutenção de grandes *sites web*, pois é possível mudar a estética de todas as páginas alterando somente um arquivo.

A denominação original traduz-se como folha de estilo em cascata, pois é possível definir vários estilos para um elemento atribuídos por prioridade de acordo com a sua origem (autor, usuário, padrão do navegador), e, ainda, por meio de vários métodos (referências externa, interna e *inline*), de modo que as propriedades

de estilo podem ser combinadas, herdadas ou sobrescritas, de um elemento-pai para um elemento-filho na estrutura da UI.

Com folhas de estilo também é possível alterar completamente o *layout* de uma página HTML, especialmente com propriedades *float*, *position*, *clear*, *width*, *z-index*, dentre outras. Os navegadores em geral não renderizam as propriedades de CSS da mesma maneira, havendo em alguns casos diferenças na disposição final dos elementos da página.

É comum a aplicação de duas folhas de estilo a uma mesma página HTML: uma para definição de propriedades de *layout* e outra para definição de propriedades visuais que não interferem no *layout*, mas apenas nas cores dos elementos e fontes, planos de fundo, imagens, etc, definindo uma espécie de estilo conhecido como *skin*, que pode ser facilmente trocado se for definido em arquivo separado. Alguns *frameworks* de componentes de UI utilizam o conceito de *skin*, como o Primefaces, pré-definindo a aparência dos componentes através de propriedades CSS.

Com a complexidade crescente das interfaces gráficas, em especial com o surgimento das *Rich Internet Applications* (RIA), houve a necessidade de desenvolvimento de tecnologias de componentes de UI para *web*, de modo a simplificar para o programador a construção da interface gráfica, incluindo marcação HTML e propriedades CSS pré-definidas e ainda códigos *Javascript* para comportamento. Algumas delas são mencionadas na seção 2.3.3, a seguir.

2.3.3 Tecnologias de componentes de UI

As tecnologias de componentes para interfaces gráficas são bastante diversificadas. Um *framework* que merece destaque é o Java Server Faces (JSF) (ORACLE, 2012), que adota o padrão MVC para aplicações web, cuja finalidade é simplificar o desenvolvimento de UI.

Extensões para RIA foram desenvolvidas por terceiros, com componentes mais ricos, como *Richfaces*, *Primefaces*, *Icefaces*, Oracle ADF. Há outros para a plataforma Java, como AWT e *Swing*, para aplicações *desktop* e MIDP para plataformas móveis. O *Microsoft Windows Presentation Foundation* (WPF) é um

framework que usa linguagem de marcação conhecida como XAML para o desenvolvimento de UI para RIA, fornecendo um modelo de programação para separação entre UI e lógica de negócios (SONNINO; SONNINO, 2012).

Esses *frameworks* tratam componentes de UI como objetos, permitindo fácil instanciação e posterior renderização para plataformas para as quais foram projetados.

Entretanto, apesar da facilidade promovida por tais *frameworks*, há alguns problemas relevantes envolvidos no desenvolvimento de UI, discutidos na seção seguinte.

2.3.4 Considerações sobre o desenvolvimento de interfaces gráficas

Do ponto de vista dos desenvolvedores, o grande problema no *design* e codificação de UI é que as linguagens de programação não foram projetadas com essa finalidade, tendo em geral apenas bibliotecas gráficas acrescentadas às suas APIs, sem, entretanto, a especificação de uma abordagem para sua geração, automática ou não, associada às características de domínios de negócio ou *workflows*. Portanto, de modo geral, linguagens de programação não são suficientes para modelagem complexa de interfaces para o usuário (KENNARD; LEANEY, 2010).

Adicionalmente, há uma grande diversidade de tecnologias e ambientes de desenvolvimento disponíveis, requerendo do desenvolvedor habilidades e conhecimentos que incluem linguagens de marcação e estilo, linguagens de programação, *frameworks* e *toolkits* para UI e adicionalmente engenharia de *software* e usabilidade.

A manutenção de UI também requer considerável esforço, tendo alguns cenários comuns, como adição de novos campos, de novas classes, alteração na ordem dos campos e novo visual que deve ser refletido em todos os campos, frequentemente com código-fonte em diversas linguagens, difícil de interpretar e, portanto, de manter (CERNY et al., 2012). É um processo trabalhoso e sujeito a erros (FALB et al., 2007), especialmente por causa da duplicação de dados de entidades de negócio através de diversas camadas do sistema, como a de persistência, a de aplicação e a de UI (KENNARD; LEANEY, 2010). Alguns

frameworks, como JPA e Hibernate, realizam um mapeamento entre a aplicação e a base de dados, permitindo definição de dados de objetos somente em código-fonte através de anotações. Entretanto, é preciso sincronizar tais informações com a camada de UI, o que se constitui ainda num esforço dobrado, susceptível a erros e inconsistências (KENNARD; LEANEY, 2011).

Portanto, em muitos trabalhos na área de geração automática de interfaces gráficas são mencionados problemas encontrados para o seu desenvolvimento considerando as técnicas atuais predominantes. Na seção seguinte, alguns trabalhos são brevemente descritos.

2.3.5 Trabalhos relacionados à geração automática de interfaces gráficas

Um gerador automático de UI constrói em tempo de execução a estrutura da interface gráfica, eliminando a necessidade de o desenvolvedor ou *designer* realizar codificação manual. A pesquisa na área de geração de interfaces gráficas é antiga e inclui diversos tipos de trabalhos, como revisões de literatura na geração automática de *layout* para apresentação da informação (LOK; FEINER, 2001), modelos que podem auxiliar na automatização de UI e, ainda, algoritmos isolados, dentre outros. Os trabalhos mais recentes foram resumidos a seguir.

2.3.5.1 OpenCTI/MedViewGen

O OpenCTI é uma iniciativa da UFPB (Universidade Federal da Paraíba), por meio do LArqSS (Laboratório de Arquitetura e Sistemas de Software), laboratório vinculado ao Departamento de Informática e da UTI (Unidade de Terapia Intensiva) do HULW (Hospital Universitário Lauro Wanderley), financiado pela FINEP (Financiadora de Estudos e Projetos), que prevê, nos seus objetivos, a construção de um RES para apoio à decisão em medicina intensiva.

A arquitetura do OpenCTI (Figura 4) foi projetada em quatro camadas: persistência/metadados, que é responsável por fazer o acesso direto à base de dados, juntamente com os metadados extraídos das especificações OWL; a camada de domínio, responsável pela gerência do ciclo de vida dos documentos e de

operações referentes ao cuidado ao paciente; a camada de aplicação, para prover a segurança das informações através dos mecanismos de autenticação e controle de acesso, além de oferecer uma arquitetura que possibilite a integração de agentes de CDS ao RES; por fim, a camada de apresentação, responsável pela geração da UI de forma estática ou dinâmica.

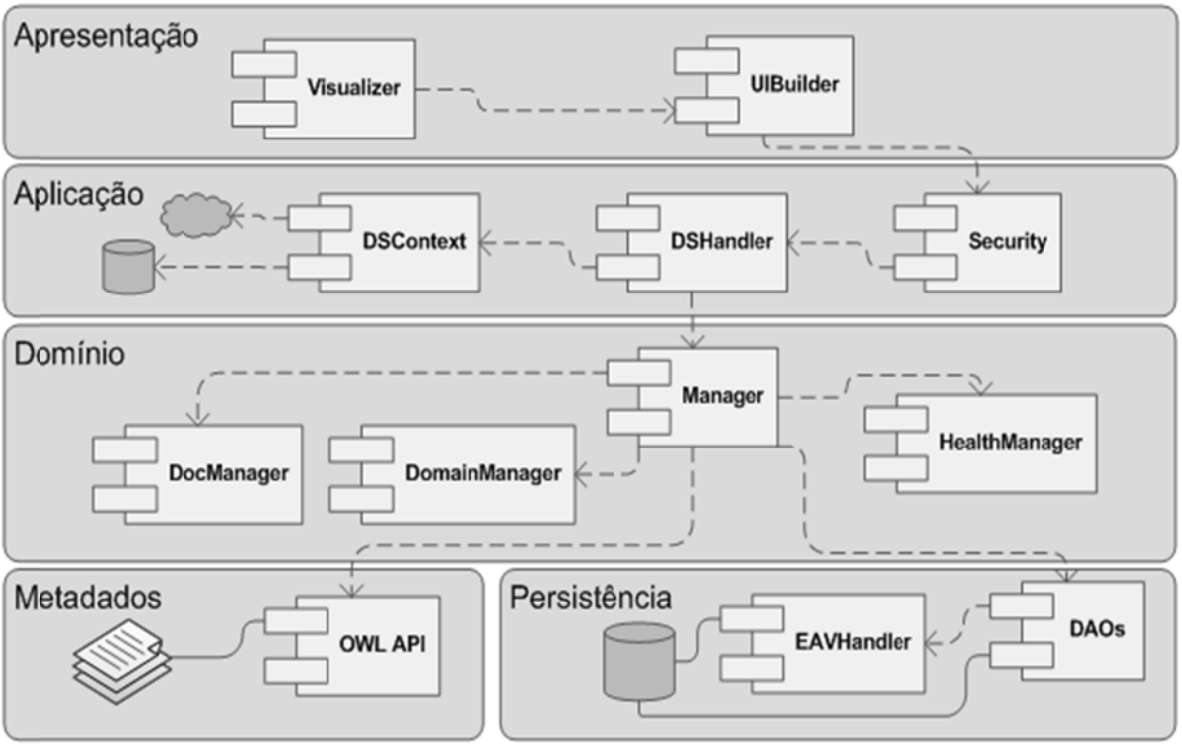

Figura 4 - Arquitetura do OpenCTI (DUARTE, 2011)

A natureza das informações clínicas não favorece o uso de uma modelagem convencional das entidades e relacionamentos em banco de dados relacionais porque esta requer o conhecimento durante o desenvolvimento dos dados que serão persistidos, seus tipos e seus relacionamentos. As informações clínicas, entretanto, são complexas, com entidades, atributos e relacionamentos variando conforme o local e com o tempo. E ainda cada instituição tem a suas próprias práticas e cultura, pacientes de diferentes categorias podem requerer informações específicas (DUARTE, 2011).

O esquema de modelagem de dados da persistência utilizado no projeto (componente *EAVHandler* da Figura 4 foi o *Entity-Attribute-Value* (EAV) que propõe uma divisão entre dados e metadados, representando cada um destes em uma

tabela distinta no banco (DUARTE, 2011). No OpenCTI, as entidades dos documentos clínicos e os atributos são representados na ontologia OWL e os dados referentes às mesmas são armazenados em uma tabela de dados principal, que se utiliza de outras tabelas.

Os documentos clínicos são, no entanto, ortogonais aos conceitos biomédicos, pois um mesmo conceito pode ser utilizado em vários documentos, através de referência, bem como alterações nos documentos não devem influenciar na base de conceitos biomédicos. Tanto os conceitos biomédicos quanto os documentos utilizados no sistema são modelados em ontologias OWL.

O *framework* diretamente responsável pela geração da UI no OpenCTI é o MedViewGen (DUARTE, 2011), sem solução, entretanto, para *layout* e adornos de UI. A API de documentos do OpenCTI gera automaticamente a interface gráfica e o código Java necessário para persistência dos dados.

2.3.5.2 Abordagem de dois modelos

O trabalho realizado por Van der Linden e colaboradores (VAN DER LINDEN et al., 2009) teve como objetivo o desenvolvimento de um *framework* para interoperabilidade no nível da apresentação da informação, entre sistemas de RES. Para isto, foi utilizada uma abordagem de dois modelos (*Two-Model Approach*), útil na separação entre o desenvolvimento da aplicação e a definição do conhecimento, o que levou os autores aplicá-la ao domínio de UI, de modo a permitir a geração de apresentações usáveis sem a necessidade de que cada estrutura de dados seja conhecida antecipadamente durante o projeto do sistema.

Os autores sugerem o uso um elemento de modelagem chamado *Display Content Unit* (DCU) para os arquétipos, que seria uma composição de um ou mais componentes, combinados com informações para orientação de exibição de conteúdo dos conceitos biomédicos descritos nos arquétipos.

A Figura 5 mostra um exemplo em XML de DCU para o arquétipo *pressão arterial* (VAN DER LINDEN et al., 2009). Há ainda, a definição de *Content Binding Unit* (CBU), que exibe lista de pares de chave-valor (*tag value,* atributos *id* e *path*) para as referências dos dados exibidos na DCU (Figura 6).

```
<contentunit id='bloodpressure' archetype='openEHR-EHR-OBSERVATION.blood_pressure.v1'>
  <hbox>
    <label id='bp_label' widgetpath='datetime/value'/>
    <value id='systolic' widgetpath='number'/>
    <text>/</text>
    <value id='diastolic' widgetpath='number'/>
    <value id='units' widgetpath='text'/>
  </hbox>
</contentunit>
```

Figura 5 - *Display Content Unit* para o arquétipo *pressão arterial* (VAN DER LINDEN et al, 2009)

```
<contentbinding id='bloodpressure' path=
'//OBSERVATION/data/HISTORY[at0001]/events/EVENT/data/ITEM_LIST[at0003]/items'/>
  <value id='bp_label' path='/POINT_EVENT[at0002]/offset/value'/>
  <value id='systolic' path='/ELEMENT[at0005]/value/magnitude'/>
  <value id='diastolic' path='/ELEMENT[at0004]/value/magnitude'/>
  <value id='units' path='/ELEMENT[at0005]/value/units'/>
</contentbinding>
```

Figura 6 - *Binding Content Unit* para o arquétipo pressão arterial (VAN DER LINDEN et al., 2009)

Entretanto, permanece em aberto a definição de uma *Content Unit Definition Language* (CUDL), que associa DCU e CBU, que seja flexível e fácil de usar para a exibição de arquétipos para o RES (VAN DER LINDEN et al., 2009).

2.3.5.3 Metawidget

O Metawidget é um gerador de interfaces gráficas com suporte a múltiplas tecnologias de UI (KENNARD; LEANEY, 2010), implementado na linguagem Java e disponível sob licença LGPL (Lesser General Public License). Os autores argumentam que qualquer gerador de UI que não se adapte às arquiteturas existentes tem pouca utilidade prática e chegaram à conclusão, através de uma pesquisa com desenvolvedores de diversas indústrias de software, de que um gerador de UI deve possuir características essenciais para sua ampla adoção e padronização.

A primeira delas é a inspeção de arquiteturas heterogêneas previamente existentes. Através da técnica conhecida como Single Source of Truth (SSOT) estrutura-se a informação de modo em que dados sobre cada elemento de um sistema são definidos e armazenados apenas em uma vez em um só local,

replicados apenas por referência, de modo que, quando há uma atualização, a mesma é propagada. Com isso, evita-se a duplicação de valores e inconsistências que podem ocorrer com a recuperação de dados de fontes distribuídas. O Metawidget cria uma SSOT temporária cujos dados serão utilizados para geração da interface gráfica por meio de um elemento chamado inspetor plugável, através de uma interface Java mínima (Inspector) e já vem com um número de implementações da mesma para extração de metadados de UI de sistemas heterogêneos. Esta técnica é conhecida como software mining. Uma limitação em aberto desta abordagem no Metawidget é a validação da completude da inspeção.

Uma segunda característica de um gerador de UI é a realização de diferentes abordagens no processamento dos resultados da inspeção da arquitetura existente. Dados brutos coletados pelo Inspector necessitam de processamento antes de sua utilização para geração de UI, como, por exemplo, critérios para ordenação e/ou exclusão de campos com base em papéis ou contexto.

Ainda, segundo os autores do Metawidget, um gerador de UI deve reconhecer e dar suporte a várias bibliotecas de widgets e misturas destas, já que há muitos frameworks de UI disponíveis em uso e, ainda, aplicar múltiplos *layouts* e mistura destes. Neste caso, o próprio Metawidget restringe os limites da geração da UI, de modo que a sua automatização refere-se a formulários (áreas em torno dos campos de entrada de dados), excluindo-se menus, navegação, aparência das páginas em si. Há várias possibilidades de organização dos componentes de UI para documentos, seja em colunas ou em uma única linha horizontal, por exemplo. O sistema permite *layouts* plugáveis, definindo, ainda, uma interface mínima com um número de implementações para dar suporte a diferentes *layouts*, baseando-se também no design pattern Decorator (GoF).

Essas características do Metawidget foram combinadas através de uma arquitetura em pipeline com elementos plugáveis, que fornece API com anotações Java próprias e reconhece outras como as do Hibernate e JSF. Alternativamente, em arquivos XML externos manualmente configurados segundo Schema XML definido pelo Metawidget, podem ser inseridas diversas configurações relacionadas à inspeção de entidades de negócio, classes de folhas de estilo (CSS) para ajuste de *layout*/estilo de interfaces web, sendo que a configuração externa parece ser a

melhor abordagem para evitar recompilação de código-fonte. Implementações personalizadas de cada elemento do pipeline também podem ser inseridas pelos desenvolvedores, através de código em linguagem Java.

2.3.5.4 Naked Objects

O *Naked Objects* foi criado por *Richard Pawson* em sua tese de doutorado (PAWSON, 2004), na Irlanda, com objetivo de promover maior entrosamento entre desenvolvedores e analistas de negócio para o desenvolvimento de aplicações baseadas em *Domain-Driven Design*[2] (DDD). É considerado tanto um padrão arquitetural quanto um *framework* de *software* (HAYWOOD, 2009).

Embora nem todos os sistemas explicitamente adotem o padrão MVC e *use-case controller pattern*[3], a maioria o faz implicitamente na forma de arquitetura de quatro camadas genéricas, esquematizada na Figura 7, à esquerda (de cima para baixo, *apresentação, controladores, objetos de domínio, persistência*), cuja relação entre as mesmas é complexa, com mapeamentos muito-para muitos em geral. Em uma implementação específica, o nome das camadas pode diferir e cada uma delas pode ser ainda subdividida em camadas adicionais, mas o conceito básico tem se tornado dominante em aplicações cliente/servidor. E ainda não oferece flexibilidade para codificação ou geração de UI (PAWSON, 2004).

[2] *Domain-Driven Design* é uma abordagem no desenvolvimento de *software* em que o foco é o domínio, em geral complexo, em que os profissionais de tecnologia da informação e analistas de negócios iterativamente refinam um modelo conceitual para resolver determinados problemas.
[3] O padrão *Use-Case Controller* lida com o problema do mapeamento das especificações de casos de uso para uma implementação de melhor custo/benefício. O padrão delega para um objeto controlador o gerenciamento do fluxo de execução de casos de uso, de modo que se mantém este conhecimento com localização definida, facilitando alterações futuras (AGUIAR et al., 2001).

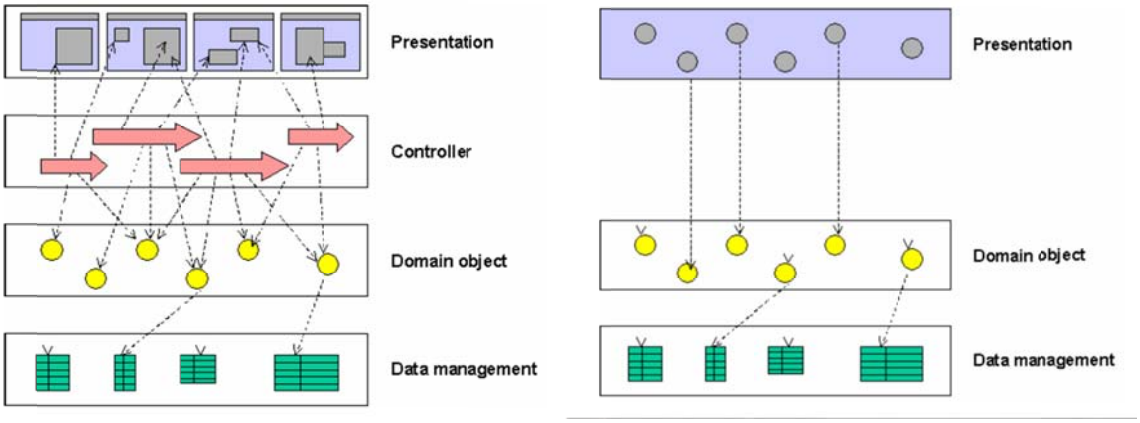

Figura 7 - Modelos de arquitetura em quatro camadas (MVC) e três camadas (Naked Objects) (PAWSON, 2004).

Na arquitetura de quatro camadas, o usuário não vê ou interage diretamente com os objetos de domínio, mas com *visualizadores* ou *controladores* que os referenciam. O conceito do *Naked Objects*, entretanto, defende uma correspondência muito forte entre essas duas camadas, de modo que se cria uma manipulação direta dos objetos de domínio, conforme Figura 7, à direita.

O *Naked Objects* defende alguns princípios:

- Toda a lógica de negócios deve ser encapsulada nos objetos de negócio.
- A UI deve ser uma representação direta dos objetos de negócio, com todas as ações do usuário consistindo em operações básicas ou invocação de métodos sobre estes objetos.

A combinação dos princípios acima forma o terceiro, que diz que a UI deve ser criada automaticamente em sua totalidade a partir da definição dos objetos de domínio, permitindo a geração de uma interface do usuário orientada a objetos (OOUI), que é genérica porque pode exibir quaisquer objetos do domínio de modo padronizado.

Algumas vantagens do padrão são:

- Ciclo de desenvolvimento mais rápido, por causa da geração automática uma OOUI, de modo que os desenvolvedores não precisam escrever

controladores ou outros objetos relacionados à camada de apresentação/UI ou persistência (HAYWOOD, 2009).

- Objetos *comportamentalmente* completos (*behaviourally-complete objects*), que significa que dentro do contexto de uma determinada aplicação, toda a funcionalidade associada a uma dada entidade é encapsulada na mesma ao invés de ser provida num controlador externo que atue sobre a entidade (PAWSON, 2004).
- Comunicação melhorada entre desenvolvedores e usuários durante o levantamento de requisitos do sistema, porque o *Naked Objects* provê uma linguagem ubíqua (PAWSON, 2004).
- Arquitetura modular que permite diferentes visualizadores, diferentes mecanismos de persistência e segurança.

O *Naked Objects* vem com dois principais visualizadores **O**OUI, um *drag-and-drop* (DnD) para aplicações em plataforma *desktop* e outro para **p**lataforma *web*.

A Figura 8 mostra um exemplo de interface gráfica gerado com o Naked Objects.

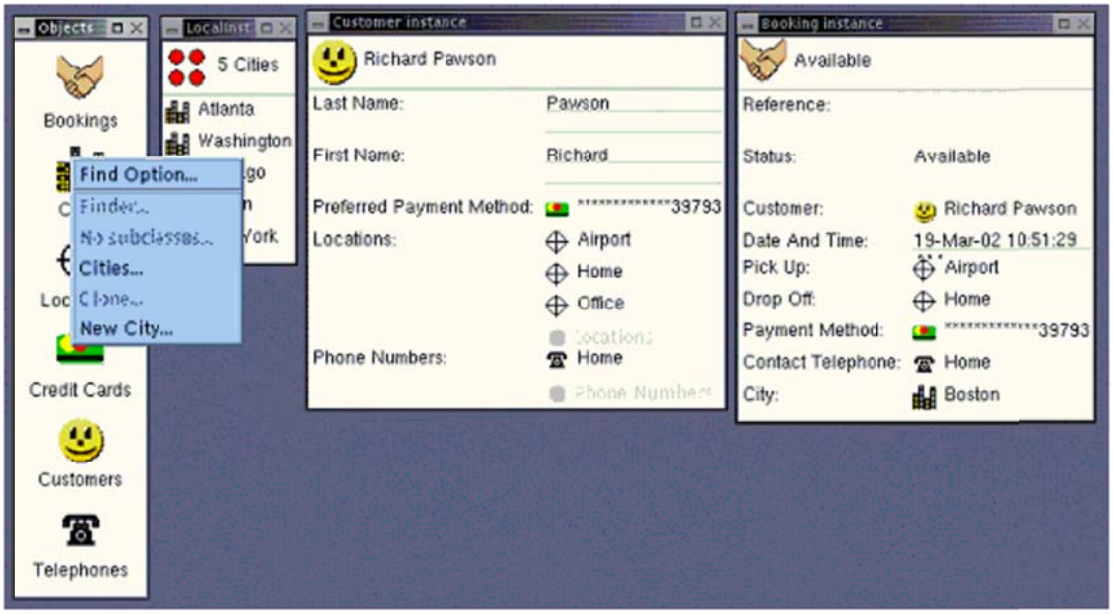

Figura 8 - Exemplo de UI gerada com Naked Objects (PAWSON, 2004).

2.3.5.5 User Interface Markup Language

UIML (*User Interface Markup Language*) é uma metalinguagem XML para especificação de interfaces para o usuário, desenvolvida por uma comissão técnica da *Organization for the Advancement of Structured Information Standards* (OASIS) e a sua última versão foi lançada em 2009 (OASIS, 2009). UIML é compatível com a especificação W3C XML 1.0.

O seu objetivo é prover uma representação canônica de qualquer UI, com mapeamento para linguagens/plataformas existentes. UIML oferece um formato único através do qual UIs podem sempre ser definidas e, adicionalmente, permite separação entre código de interface gráfica e código de lógica de negócios, sendo extensível para futuras tecnologias de interface gráfica. A UIML é uma especificação aberta e, portanto, não requer licença para ser implementada.

2.4 Considerações finais

Os trabalhos encontrados mostraram diferentes escopos na geração automática de UI. Alguns, como o *Naked Objects* (PAWSON, 2004), geram a interface da aplicação completa, outros, a interface de formulários, como o *Metawidget* (KENNARD; LEANEY, 2010), o *MedViewGen* (DUARTE, 2011), o trabalho descrito por Cerny e colaboradores (CERNY et al., 2012), a geração de formulários em aplicações de pesquisa de campo em ecologia (JONES et al., 2007) e abordagem de dois modelos (VAN DER LINDEN et al., 2009) para sistemas RES. Alguns ainda alegam gerar não somente a UI, mas o código da aplicação necessário para persistência dos dados (DUARTE, 2011) (TORTOSA et al., 2012), outros permitem incorporação de validadores e outros *scripts* em componentes de UI (KENNARD; LEANEY, 2010), outros, entretanto, não especificaram sobre esta característica.

Ainda, foram encontrados trabalhos relacionados a especificações de modelos de interfaces gráficas, a exemplo de UIML (OASIS, 2009), Mozilla XUL (*User Interface Language*) (MOZILLA DEVELOPER NETWORK, 2005) e WebForms (W3C WORLD WIDE WEB CONSORTIUM, 2011), entretanto, nenhuma com ampla adoção pela indústria de *software*.

O trabalho descrito por Hervás (HERVÁS; BRAVO, 2011) mostrou-se bastante modular. Através de um framework, interfaces gráficas para vários dispositivos são geradas dinamicamente com base em informações de contexto, serviços, padrões de projeto em UI e tarefas de usuário, descritos em ontologias. No entanto, para definição de *layout*/estilo de UI em menor nível de abstração, como posicionamento e dimensões de componentes de UI, não foram fornecidos detalhes adicionais de técnicas usadas para computação dos mosaicos aplicados nas diversas visualizações de interfaces gráficas do trabalho.

Para muitos autores, a maior preocupação motivadora para geração de UI automática é do ponto de vista dos desenvolvedores, que devem executar tarefas desnecessárias para modelagem/codificação de interfaces gráficas (CERNY et al., 2012) (KENNARD; LEANEY, 2010), perdendo o foco de seu trabalho como programadores que seria resolver problemas de domínio de negócios. Entretanto, para que um gerador de UI seja adotado pelos desenvolvedores, ele deve satisfazer algumas características, especialmente se adaptar a arquiteturas existentes (KENNARD; LEANEY, 2011). Do ponto de vista do usuário, para uma adoção satisfatória de um sistema com UI gerada automaticamente, a mesma deve ser bem projetada e oferecer um *layout* adequado que não prejudique usabilidade nem a estética da interface gráfica.

Considerando que a área médica é um dos domínios mais complexos para modelagem, os trabalhos de geração automática de RES específicos mostram maior preocupação em separar o conhecimento da apresentação, seja com uso de arquétipos ou ontologias, em relação àqueles voltados para nenhum domínio específico.

Neste capítulo, portanto, foram apresentados os conceitos e tecnologias principais necessários ao trabalho proposto, como noções da estruturação de *templates* de documentos clínicos e conceitos de interface gráfica e *layout*. Além disto, foram brevemente descritos alguns trabalhos sobre geração automática de UI já publicados, fornecendo uma visão do estado atual das áreas principais relacionadas ao trabalho proposto e demonstrando a existência de questões a serem exploradas.

3

Modelos ontológicos para geração de layout de UI

Uma das questões práticas mais importantes num sistema de *layout* automatizado baseado em *restrições* é onde obtê-las (LOK; FEINER, 2001). Para isto, os modelos ontológicos do OpenCTI inicialmente desenvolvidos em dois trabalhos pesquisados (DUARTE, 2011) (NÓBREGA, 2010) foram estendidos para abranger as alterações necessárias para implementação de *layout*/estilo de UI, bem como formatos de apresentação específicos para alguns arquétipos em medicina.

Os modelos ontológicos foram representados em diagramas de classes em Unified Modeling Language, UML (OBJECT MANAGEMENT GROUP, 2011). Na seção 3.1, *Modelo ontológico de conceitos biomédicos*, relatamos as alterações realizadas na ontologia, em especial aqueles necessários para estruturação do *layout*. Na seção 3.2, *Modelo ontológico de documentos clínicos*, foram descritos os formatos de apresentação para dados médicos e sua modelagem ontológica, com base em observações de interfaces gráficas em saúde e representações de dados comuns na área presentes na literatura e em documentos clínicos. Na seção 3.3, *Modelo ontológico de Componentes/UI*, o modelo ontológico de componentes foi estendido para englobar propriedades de *layout* e estilo gerais, aplicáveis a documentos de qualquer área e aos próprios componentes. Por fim, na seção 3.4, *Modelo ontológico para mapeamento de* tecnologias de UI*,* foi realizada uma modelagem adicional para mapeamento de tecnologia de *layout*/estilo de interfaces web, conhecida como *Cascade Style Sheet* (CSS).

Em cada seção também são descritas observações importantes relacionadas à modelagem e implementações de algoritmos ou alterações de algoritmos existentes no trabalho anterior usado como base, o MedViewGen (Duarte, 2011),

tanto para conversão do modelo ontológico em árvore genérica de componentes de interface gráfica e sua conversão para as tecnologias utilizadas, JSF e neste trabalho CSS. Para edição dos arquivos OWL das ontologias usadas no presente estudo, foi usado o *software* Protégé, brevemente descrito na seção 2.2.1. Por simplicidade, optamos por inserir ao longo do trabalho capturas de tela do *Protégé* para as definições de classes ontológicas criadas ou modificadas, com a devida referência.

3.1 Modelo ontológico de conceitos biomédicos

Ontologias biomédicas e *templates* de documentos clínicos bem modelados podem facilitar o trabalho do gerador de *layout*, portanto, houve a necessidade de remodelar parcialmente as ontologias já implementadas no projeto OpenCTI, usado no presente trabalho, para alcançar os objetivos propostos.

A Figura 9 exibe a representação do modelo de conceitos biomédicos desenvolvido para o projeto OpenCTI e em uso no presente estudo.

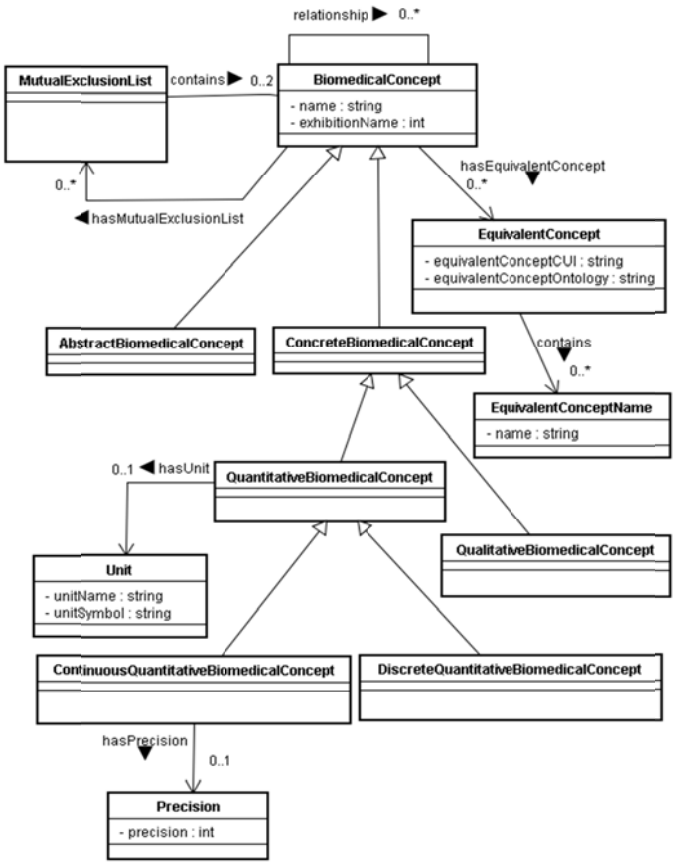

Figura 9 - Modelo ontológico de conceitos biomédicos (NOBREGA, 2010).

O modelo ontológico de conceitos biomédicos foi desenvolvido no trabalho de Nóbrega (NÓBREGA, 2010). Uma pequena modificação foi realizada para o presente estudo através da inclusão de um *dataProperty*, para determinar, ainda que arbitrariamente, valores máximos em termos numéricos que o conceito suporta e número máximo de caracteres em texto livre para conceitos com valores textuais longos, como campo para observações. Essa alteração auxilia na determinação de uma restrição no *layout* dos componentes de interface para entrada de dados, pois fornece um metadado para o cálculo aproximado do tamanho máximo do componente baseado no tamanho máximo do dado por ele exibido. Por exemplo, o valor máximo do conceito biomédico quantitativo *temperatura corporal* foi arbitrariamente determinado igual a 45 °C, o que nos dá um tamanho máximo de um componente de entrada de texto no valor de quatro caracteres, considerando uma precisão de uma casa decimal, mais o separador (ponto ou vírgula).

Há conceitos que, entretanto, admitem valores máximos definidos, como alguns parâmetros de oxigenação para programação de ventilador mecânico, e foram inseridos na ontologia de conceitos biomédicos.

A Figura 10 exibe a definição da classe *QuantitativeBiomedicalConcept*, onde foi realizada a alteração pela adição do *dataProperty maxValue* (valor máximo).

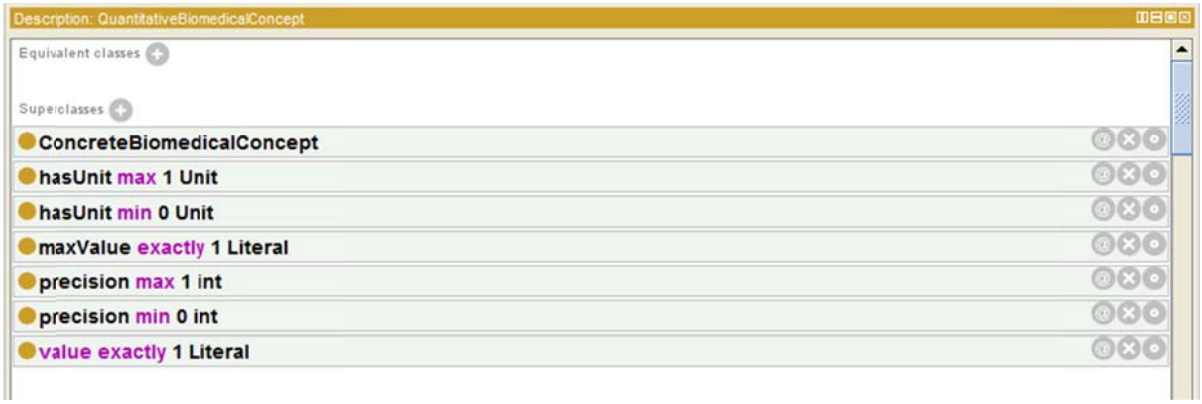

Figura 10 - Definição da classe QuantitativeBiomedicalConcept

Outros exemplos são mostrados, conforme o Quadro 1:

Conceito biomédico	Valor máximo arbitrário	Precisão
Temperatura corporal	45	0,1
Frequência cardíaca	300	0
Pressão arterial sistólica	300	0,1
PEEP (Positive End Expiratory Pressure, pressão expiratória positiva final) (como parâmetro para ventilação mecânica)	50	-
Pressão controlada acima da PEEP (como parâmetro para ventilação mecânica)	100	-
Frequência respiratória (como parâmetro para ventilação mecânica)	100	-

Quadro 1 - Valores máximos arbitrários para conceitos biomédicos

De todos os modelos ontológicos utilizados neste trabalho, o de Conceitos Biomédicos ainda é o que permanece mais homogêneo, abrangendo entidades relacionadas apenas à área médica.

3.2 Modelo ontológico de documentos clínicos

Os modelos ontológicos de conceitos biomédicos e de documentos clínicos se relacionam originalmente através da Classe *Archetype*, que é a unidade de informação do documento contendo o conceito biomédico. Esta é a estrutura do documento que comporta dados de acordo com um determinado metamodelo semântico.

O modelo ontológico de documentos clínicos em uso no projeto OpenCTI foi desenvolvido por (DUARTE, 2011), Figura 47 (Anexo I), e está sendo utilizado no presente trabalho como base, com alterações inseridas neste estudo e apropriadamente indicadas, resultando no modelo da Figura 11.

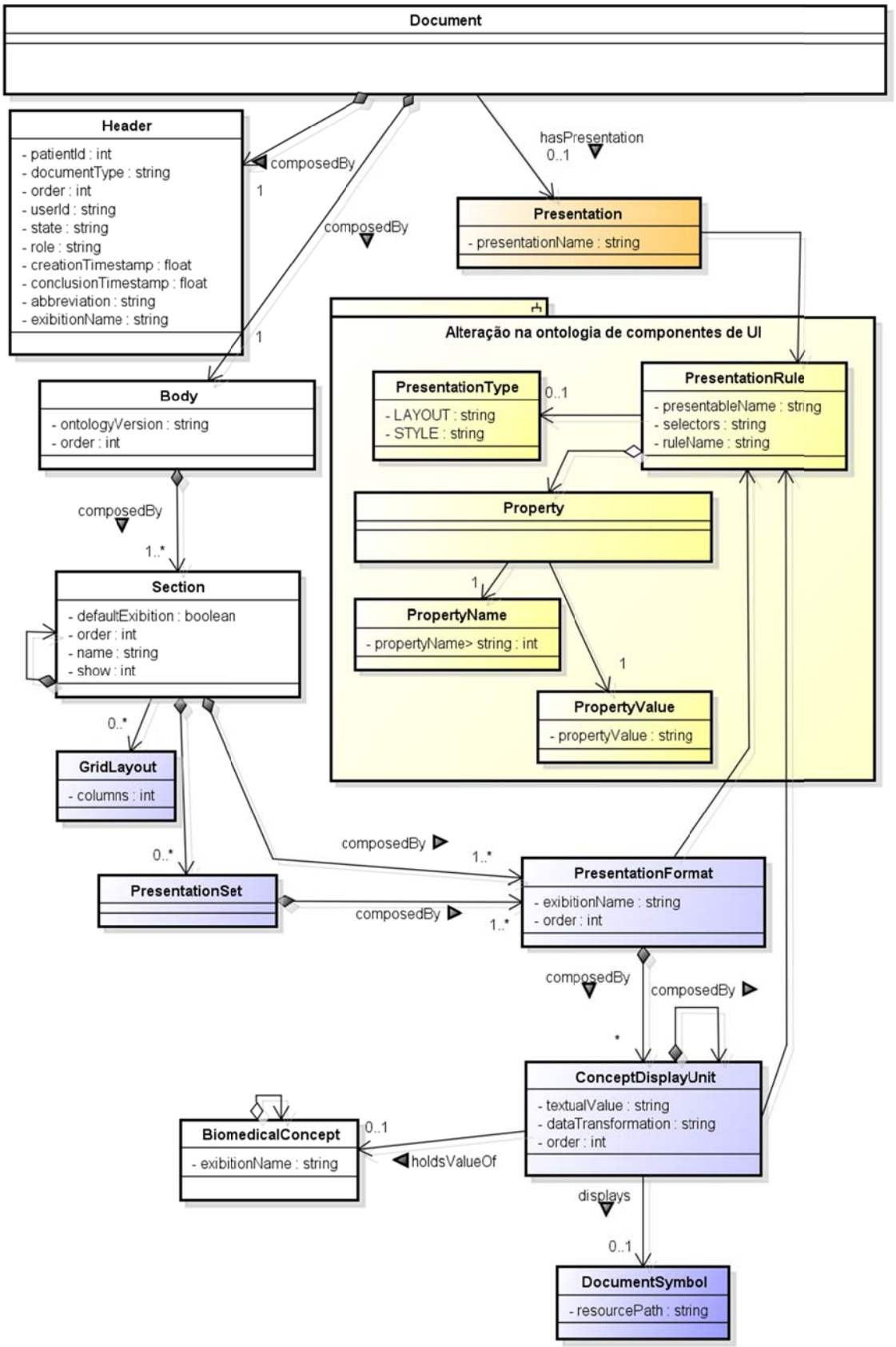

Figura 11 – Modelo ontológico de documentos adaptado do MedViewGen (DUARTE, 2011)

Novas classes representadas em cores na Figura 11 foram acrescentadas ao modelo original para acomodar arquétipos que requerem *layouts* especiais para área médica, que chamamos neste trabalho de *PresentationFormat*. As alterações realizadas no modelo de documentos estão relacionadas com o modelo de componentes genéricos de UI e com o modelo de conceitos biomédicos, conferindo a este modelo um grau de heterogeneidade bastante elevado.

Verifica-se, portanto, que a apresentação da informação é uma área multidisciplinar, sendo, portanto, difícil a construção de uma ontologia para sua representação (HERVÁS; BRAVO, 2011).

Todos os itens adicionais e observações sobre a modelagem e implementação de algoritmo foram descritos nas subseções seguintes.

3.2.1 PresentationFormat

A classe *PresentationFormat* (formatos de apresentação) (Figura 12) desenvolvida no presente estudo representa a definição da representação visual de um conceito biomédico, com restrições espaciais oriundas de convenções de *layouts* informais da área médica, reorganizando a disposição e/ou tamanho e estilo dos elementos visuais, para melhorar ou tornar adequada a exibição da informação.

Optamos por usar tais formatos na ontologia de documentos clínicos em vez de naquela de conceitos biomédicos porque tais conceitos existem sem *layouts* ou informações de estilo, mas sua apresentação nos documentos do RES ou mesmo em formulários em papel é que requer convenções espaciais e padronizações visuais bem definidas e consistentes.

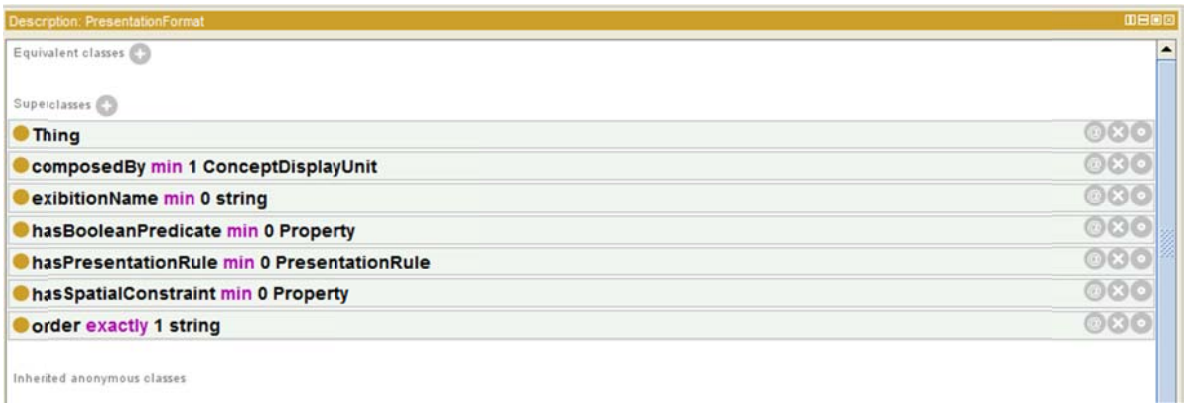

Figura 12 - Definição da classe PresentationFormat

Alguns atributos e relacionamentos dos modelos ontológicos foram aproveitados do trabalho de (DUARTE, 2011). A classe *PresentationFormat* possui um atributo *exibitionName* (*string*) que refere-se ao seu nome de tela, para o usuário, quando necessário. Outro atributo é o *order* (*int*), que representa a ordem em que o formato aparece na seção do documento, sendo, portanto, obrigatório, pois representa uma restrição para o *layout*.

Um *PresentationFormat* não se relaciona diretamente com a classe *Archetype,* tendo sido modelado como um *wrapper* para o conceito biomédico/arquétipo que formata, juntamente com a classe *ConceptDisplayUnit* (descrita na seção 3.2.2).

A justificativa para o não uso dos arquétipos já instanciados no modelo ontológico de documentos em associação com *PresentationFormat* é que a modelagem de exibição de conteúdo é complexa, cada item de um conceito biomédico pode ter suas regras próprias de *layout* e estilo, restrições de dados e, considerando que cada conceito biomédico pode ter tantos outros como parte ou composição, cada unidade mínima de informação apresentável deve ser considerada na modelagem, seja ela de dados do usuário, rótulos, símbolos especiais, unidades de medição, ícones, imagens, texto de ajuda, etc.

Portanto, cada arquétipo pode ser reconstruído com informações de *layout* e estilo para ser considerado da classe *PresentationFormat*, do contrário, ele terá apenas o *layout* comum possível de acordo com a implementação original do algoritmo de atribuição automática de componentes (AAC), apresentado no Capítulo

4, e herdará, possivelmente, o estilo definido na *Presentation* do documento ao qual pertence (descrita na seção 3.2.3).

Os relacionamentos (*objectProperties*) (Figura 12) desta classe são opcionais e estão descritos a seguir:

- *hasPresentationRule* – para indicar que o indivíduo pode ter opcionalmente regras definidas para *layout* e estilo, que serão mapeadas e traduzidas para a tecnologia de interface utilizada, CSS. O objeto deste relacionamento é um indivíduo da classe *PresentationRule* (descrita na seção 3.3.1).
- *hasSpatialConstraint* – para indicar que o indivíduo pode ter restrições espaciais definidas para auxiliar no algoritmo de AAC. O objeto deste relacionamento é um indivíduo da classe *Property* (descrita na seção 3.3.2), propriedade do tipo chave = valor (exemplo: orientação = horizontal).
- *hasBooleanPredicate* - para indicar que o indivíduo pode ter algum predicado booleano, o que é na verdade numa especialização de restrições, sendo neste caso somente aquelas do tipo booleanas. Optamos neste caso por separar *restrições espaciais* e *predicados booleanos* em dois relacionamentos por melhor aplicabilidade ao algoritmo já implementado, mas nada impede que um predicado booleano imponha uma restrição espacial.

Como exemplo de *PresentationFormat*, utilizaremos deste ponto em diante o formato para o conceito biomédico de pressão arterial, composto de dois conceitos mais simples, pressão arterial sistólica e pressão arterial diastólica, um separador, um título, um ícone e uma unidade de medição (mmHg), conforme o resultado exibido na Figura 13.

Figura 13 - Formato de apresentação para o conceito pressão arterial

Outros exemplos de formatos de apresentação obtidos foram para os conceitos de temperatura corporal e de frequência cardíacas, análogos ao anterior, conforme ilustrados na Figura 14.

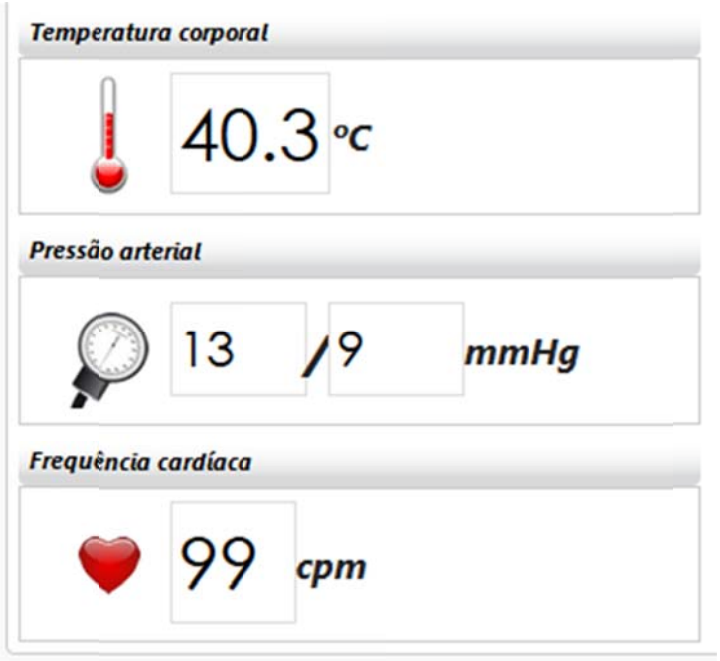

Figura 14 – Outros formatos de apresentação para conceitos biomédicos

Na seção a seguir, descrevemos a classe *ConceptDisplayUnit* concebida para a modelagem de *PresentationFormat*.

3.2.2 ConceptDisplayUnit

Para compor o *layout* dos conceitos ontologicamente, dividimos cada unidade de informação a ser exibida e consideramo-la uma *ConceptDisplayUnit* (Figura 11) que corresponde, teoricamente, à menor unidade de exibição de dados de um conceito biomédico na tela do computador.

Além da própria estrutura da UI, representada pelos componentes, ela pode conter também informações de várias fontes distintas e propósitos diversos que podem ser apresentados e, portanto, representados por uma *ConceptDisplayUnit*. Adiante, apresentamos uma lista dos tipos de conteúdo de UI identificados no presente trabalho, a saber:

- Conteúdo de UI, que são os rótulos dos conceitos e seções do documento, cabeçalhos, unidades de medida, itens de ajuda, ícones, mensagens de validação de dados etc.
- Conteúdo de usuário, que são os dados inseridos no sistema.
- Conteúdo calculado, que resulta de processamento de dados do usuário ou de outras fontes de dados pelo sistema e inserido dinamicamente na UI.

No exemplo utilizado de *PresentationFormat* para o conceito biomédico de pressão arterial, identificamos os seguintes microconteúdos a serem exibidos, listados a seguir.

- Um possível rótulo, textual ou imagem, que seria o valor do *dataProperty* exibitionName do *PresentationFormat* ou conceito ou ainda um ícone equivalente.
- O valor da pressão arterial sistólica do paciente, dividido por dez.
- A barra que separa os dois valores
- O valor da pressão arterial diastólica do paciente, dividido por dez.
- O símbolo da unidade de medição de cada pressão (mmHg) ainda que opcional neste caso.

Visualizando o resultado esperado, verifica-se que há conteúdos atômicos distribuídos na tela, como na Figura 15:

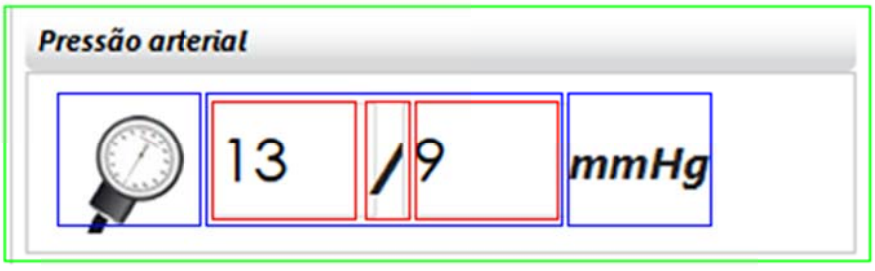

Figura 15 - Conteúdos atômicos para o conceito de pressão arterial

Na Figura 15, observamos as unidades marcadas em vermelho como conteúdo atômico, identificando em cada uma delas uma *ConceptDisplayUnit*, no mesmo nível de aninhamento, mas que devem permanecer espacialmente próximas e formatadas uniformemente em termos de tipografia (tipo, tamanho e cor de fonte),

por exemplo, o que nos leva a identificação de outra unidade agrupadora, em um nível superior, marcada em azul. Identificamos, neste caso, uma composição entre indivíduos da mesma classe, semelhante ao *design pattern composite*. Temos, portanto, uma composição onde cada item é um *ConceptDisplayUnit*. Nesse mesmo nível, à esquerda, um rótulo para o conceito representado por um ícone e, à direita, a unidade de medição. No nível raiz, temos o PresentationFormat, marcado em verde, representado pelo painel agrupador e pelo título que é o nome do conceito.

A modelagem final da classe *ConceptDisplayUnit* está ilustrada na Figura 16 (captura de tela do Protégé).

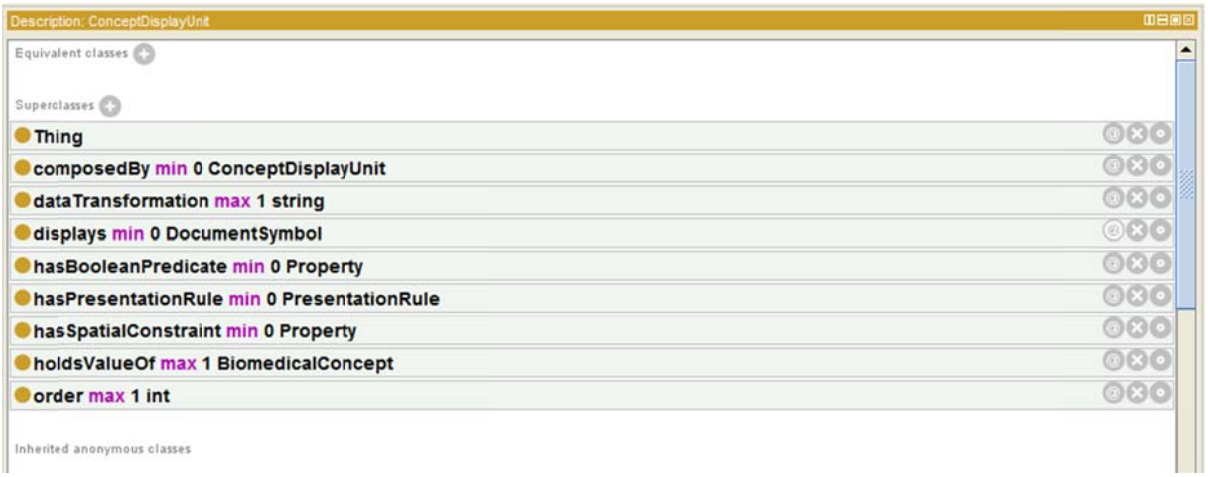

Figura 16 – Definição da classe ConceptDisplayUnit

Os relacionamentos desta classe estão comentados nos tópicos seguintes:

- *composedBy ConceptDisplayUnit* – relacionamento opcional que indica que a composição entre indivíduos da mesma classe.

- *order*, do tipo inteiro, refere-se à ordem sequencial do conteúdo a ser exibido, caso haja mais de um elemento *ConceptDisplayUnit* no indivíduo. A intenção é que a sequência visual seja preservada. Pode ser considerada uma restrição para o *layout*.

- *displays* – o objeto deste relacionamento pertence à classe *DocumentSymbol* (seção 3.2.3) , que se refere a um recurso, identificável por uma URL, que deva ser exibido, seja como uma imagem, ícone, arquivo de som, vídeo, caractere especial, etc., em termos de componentes genéricos mapeáveis, que pudesse estar relacionado ao conceito e seu formato de apresentação

dentro do documento em construção. No caso de um modelo para frequência cardíaca, pode ser usado um ícone ❤ em vez do rótulo "*Frequência cardíaca",* útil para exibição em dispositivos com tamanho de tela reduzido.

- *holdsValueOf BiomedicalConcept* – relacionamento que sinaliza a exibição de algum atributo de conceito biomédico para entrada de dado (ou valor).

- *dataTransformation*, do tipo *string*, serve como identificador da transformação necessária sobre o dado exibido para que o algoritmo AAC a realize sobre o valor do conceito biomédico que a compõe*,* que, no caso do exemplo em análise, seriam os valores das pressões sistólicas e diastólicas que são medidos em *mmHg*, com valores normais de 120 mmHg para a sistólica e 80 mmHg para a diastólica, mas que são exibidos neste formato de apresentação como 12/8, requerendo uma divisão por dez de cada um, caso tenham sido persistidos no formato de valor original. Usamos um atributo *string* apenas para identificar o tipo de transformação, já que uma modelagem matemática que permita uma referência para um indivíduo de tipo *MathematicalCalculation*, por exemplo, está fora do escopo deste trabalho. Outros tipos de transformações de dados poderiam ser necessários, tais como abreviações, truncamentos, caixa alta, caixa baixa, aplicação de máscara formatadora, etc., desde que padronizados ou compreensíveis pelos usuários finais do sistema.

- *hasPresentationRule,* opcional, para informações adicionais de *layout* e estilo que sejam específicas ao microconteúdo apresentado.

O *ConceptDisplayUnit* tem ainda outros dois relacionamentos, *hasSpatialConstraint* e *hasBooleanPredicate*, também presentes na classe *PresentationFormat*, tendo a mesma finalidade em ambas.

Considerando a necessidade de exibição de símbolos (ícones, caracteres especiais), optamos por definir uma classe acessória chamada *DocumentSymbol*, descrita na seção seguinte, 3.2.3.

3.2.3 DocumentSymbol

A classe *DocumentSymbol* (Figura 17) foi criada para permitir a representação de informação visual inerente à interface gráfica do documento, respeitando a semântica do(s) conceito(s) formatado(s). Um item deste tipo pode representar um rótulo em forma de imagem, um ícone qualquer, um caractere especial. Componentes visuais podem ser úteis para representar informação de forma resumida, prescindindo muitas vezes da necessidade de mensagens textuais.

Como a exibição de tais itens requer elementos diferentes na interface final para o usuário, optamos por incluir na ontologia, componentes genéricos distintos para itens do tipo ícone e itens do tipo texto. Neste caso, para representar um *DocumentSymbol*, instanciamos na ontologia respectiva a definição de um componente genérico chamado *image* para símbolos que referenciem algum arquivo de imagem e usamos um componente do tipo *outputText*, para símbolos representáveis por caracteres especiais, como aqueles presentes em alguns tipos de fontes, caracteres separadores, etc.

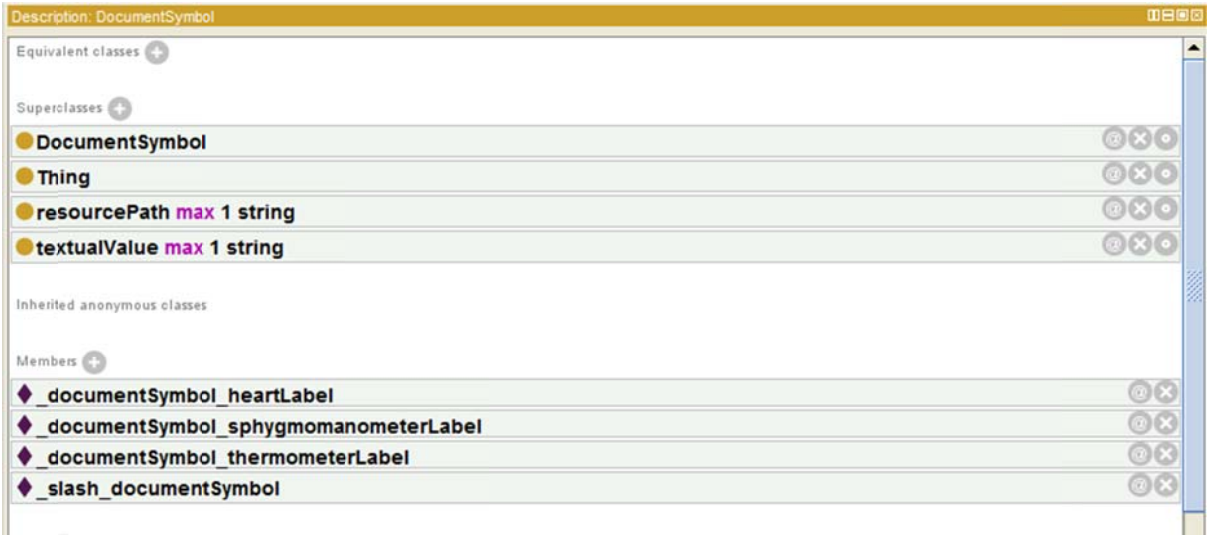

Figura 17 - Definição da classe DocumentSymbol.

O atributo *resourcePath* refere-se ao caminho do arquivo de imagem (URL) que deve ser exibido no documento. Para os exemplos deste trabalho, usamos imagens do tipo PNG (*Portable Network Graphics*) de tamanho pré-definidos para representar os ícones correspondentes a cada conceito formatado em cada indivíduo *PresentationFormat*. Em termos de codificação, se não estiver presente o

valor deste atributo, o algoritmo implementado procura pelo atributo *textualValue* que representa algum valor em caracteres que deve ser exibido no local indicado. Apenas um atributo é obrigatório, o que nos levaria a uma definição mais exata da classe como *DocumentSymbol hasValue exactly 1 (resourcePath OR textualValue)* .

Para o formato do conceito de pressão arterial, usamos dois indivíduos *DocumentSymbol*: o primeiro, para o ícone, o segundo, para a barra que separa os dois valores, pressão sistólica e pressão diastólica.

A Figura 18 exemplifica a relação entre os indivíduos instanciados nas ontologias de conceitos biomédicos e documentos para a criação do indivíduo da classe *PresentationFormat* representado pelo resultado visual da Figura 13, para pressão arterial.

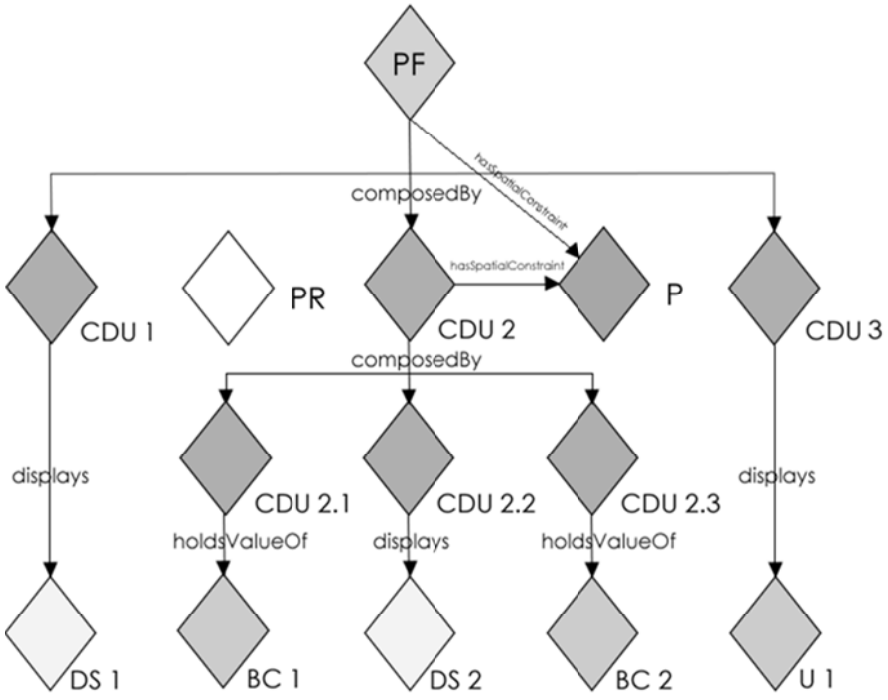

Figura 18 - Relacionamentos entre indivíduos do formato de apresentação para pressão arterial

Cada indivíduo da Figura 18, representados por losangos, foi instanciado nas ontologias a que pertencem. Como a ontologia de documentos importa a ontologia de conceitos biomédicos e a de componentes de UI, é possível criar um novo

indivíduo que se relaciona com indivíduos de várias origens. Portanto, participam da composição do formato de apresentação correspondente à Figura 13 os seguintes indivíduos:

- Um do tipo *PresentationFormat* (indicado como PF);
- Seis do tipo *ConceptDisplayUnit* (indicados como CDU);
- Dois do tipo *DocumentSymbol* (DS1 para o ícone, DS2 para a barra);
- Dois indivíduos do tipo *QuantitativeBiomedicalConcept* (BC 1 para pressão sistólica e BC 2 para diastólica) e um indivíduo *Unit* (indicado como U 1) para a unidade *mmHg*;
- Um do tipo *PresentationRule* (PR), neste caso, para definição do tamanho da fonte e foi reusado por todos os indivíduos CDU que se relacionam diretamente com ele, mas foram omitidas as linhas dos relacionamentos entre eles para melhor legibilidade;
- Um do tipo *Property* (P) para definição da restrição espacial (*hasSpatialConstraint*) na horizontal dos CDU dentro do PresentationFormat que também foi reusado para o *ConceptDisplayUnit* (CDU 2, Figura 18), que possui filhos que devem estar dispostos na horizontal. Isto é importante para orientação da inserção do *grid* pelo algoritmo implementado no presente estudo.

Portanto, deste modo são estabelecidos os relacionamentos entre os indivíduos das novas classes adicionadas no presente estudo para a ontologia de documentos. Os *PresentationFormat* representam pequenos *layouts* locais num formulário definidos para a apresentação de informação em formato específico. Na seção seguinte, veremos a descrição de outra classe criada para a ontologia de documentos.

3.2.4 Presentation

A classe *Presentation* (Figura 19) foi criada no modelo de documentos clínicos para incluir metadados referentes à apresentação do tipo *layout* e estilo separadamente, que incluem propriedades do tipo chave=valor, representados, no

caso, pela classe *PresentationRule,* incluída em outro modelo ontológico, o componentes de UI, descrito adiante na seção 3.3.1.

O *dataProperty presentationName* deve ser único, permitindo a identificação da apresentação do documento dentro da ontologia.

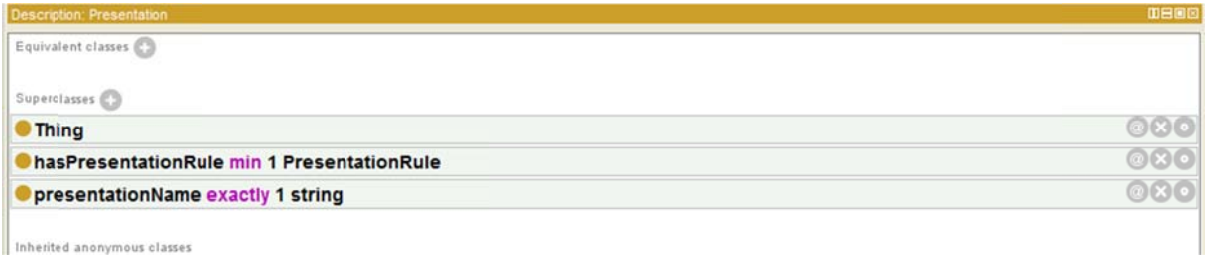

Figura 19 – Definição da classe Presentation

A recomendação é que cada instância de documento tenha sua *Presentation* com *PresentationRule*s criadas de modo a representarem separadamente propriedades de *layout* e de estilo. O relacionamento *hasPresentationRule* vincula ao menos uma *PresentationRule* (min 1) para o documento, ou seja, ao menos um grupo de propriedades para *layout* ou para estilo, qual houver. Essa separação é conveniente especialmente para manutenção da aplicação, pois é comum que interfaces gráficas em geral tenham sua aparência alterada parcialmente, seja quanto ao *layout* somente, ou quanto ao estilo somente. Agrupando isoladamente tais propriedades com propósitos diferentes, é mais fácil alterar um dos itens independentemente e mudar sem grandes esforços o estilo padrão de um documento para outro.

Consideramos aqui tais propriedades como metadados da UI pelo fato de não atribuírem novos nós visuais à árvore genérica de componentes criada pelo algoritmo AAC, pois são apenas informações que alteram a aparência dos componentes de UI já existentes. Portanto, indivíduos da classe *Presentation,* ainda que estejam presentes nos documentos, não fazem parte do conceito da classe *DomainElement* estabelecido na originalmente ontologia de componentes de UI, tendo sido, por isso, excluídos da atribuição automática de componentes no presente estudo.

O processamento em *software* da classe *Presentation* e seus relacionamentos com outros indivíduos é realizado de modo a inserir um objeto com os metadados de apresentação no nó raiz da árvore genérica de UI gerada.

Outras classes inseridas na ontologia, como a *PresentationFormat*, já fornecem orientações para disposição espacial de itens específicos na tela, sendo, portanto, diretamente responsáveis pelo *layout* dos elementos de UI relacionados aos conceitos biomédicos que formata, podendo definir novos componentes na estrutura da árvore genérica.

Por fim, através do relacionamento *hasPresentation* entre uma instância da classe *Document* e de *Presentation*, vincula-se a apresentação ao documento, conforme Figura 20:

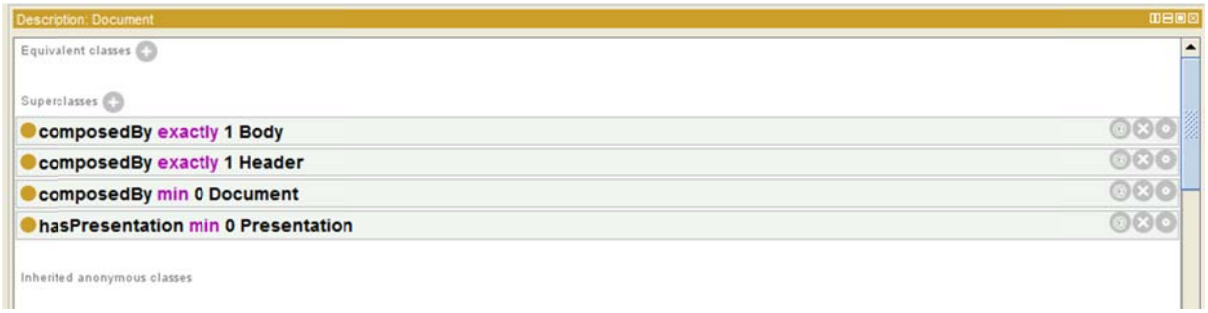

Figura 20 –Redefinição da classe Document

3.2.5 GridLayout e PresentationSet

Foram criadas classes acessórias para auxiliar na definição de *grids* para distribuição dos componentes na seção do documento. Através da classe *GridLayout*, Figura 21, atribui-se o número de colunas padrão para o *grid* de cada seção do documento.

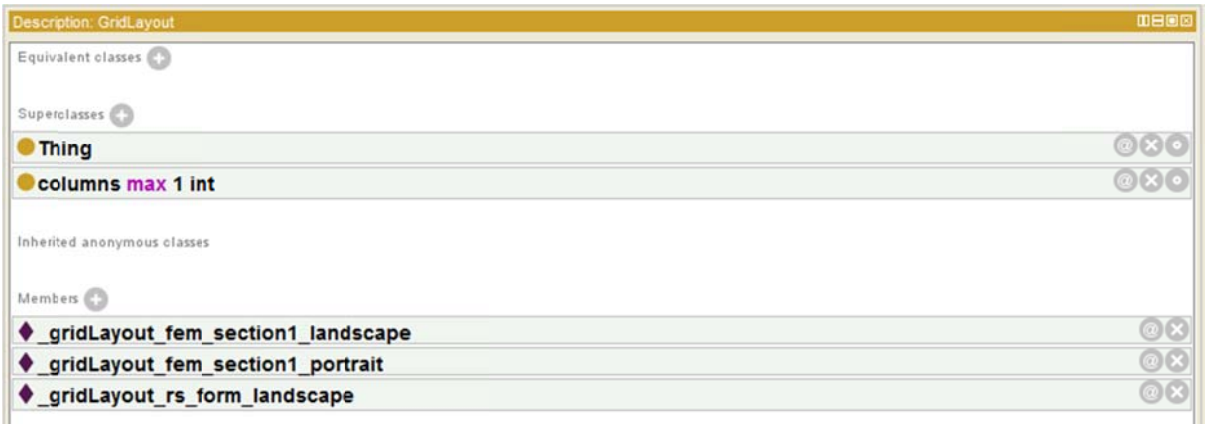

Figura 21 - Definição da classe GridLayout

Considerando também a necessidade de trabalhar com grids esparsos, em que não há em todas as linhas um mesmo número de colunas, foi criada a classe *PresentationSet*, Figura 22, definida como uma composição de itens do tipo *PresentationFormat* que devem ser exibidos do início ao fim de uma linha na seção do documento. Para a mesma foi definido o atributo *order*, inteiro, para determinar a ordem deste na seção do documento, do mesmo modo que na classe *PresentationFormat*. Entretanto, o número de relacionamentos *composedBy PresentationFormat* indica a quantidade de colunas (*gridCell*) necessárias para preencher esta linha. A seção do documento, portanto, pode ser composta de *PresentationFormat* e de *PresentationSet*.

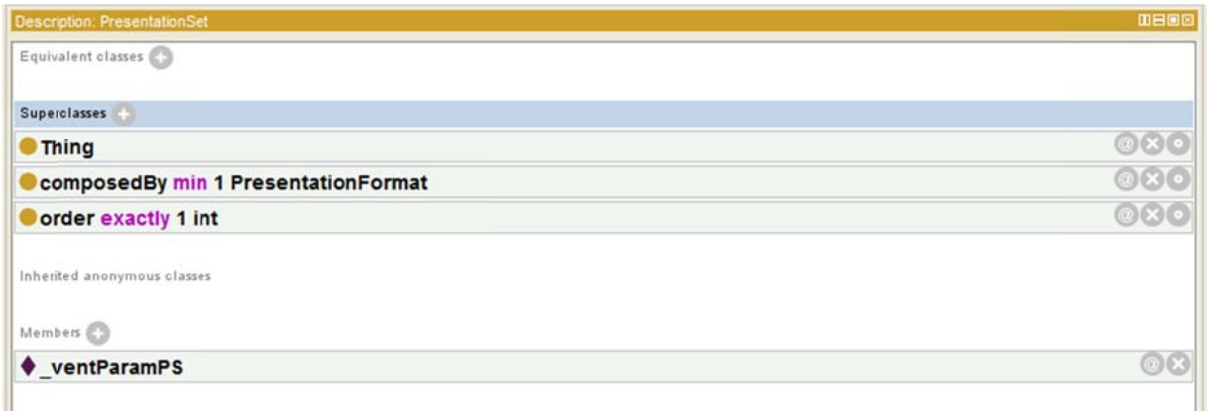

Figura 22 - Definição da classe PresentationSet

Aqui se encerram as definições de classes para o modelo ontológico de documentos, que pode ser considerado, portanto, o mais heterogêneo de todos

aqueles que fizeram parte deste estudo, pois se utiliza de classes e indivíduos do modelo de conceitos biomédicos e da ontologia de componentes genéricos, além de suas próprias definições. Isso pode ser justificado pelo fato de em cada documento estar estruturada toda a informação necessária para o usuário, sendo o formulário eletrônico que contém o documento a interface final entre o usuário (profissional de saúde, no caso) e o próprio sistema.

3.3 Modelo ontológico de Componentes/UI

O modelo ontológico de interface gráfica proposto no trabalho de (DUARTE, 2011) foi remodelado no presente estudo para abranger, não somente componentes, mas também metadados para *layout* e restrições, estilo e propriedades relacionadas aplicáveis aos componentes.

Neste trabalho, não usamos eventos de UI e sistemas de CDS. O diagrama da Figura 23 ilustra o modelo de componentes de UI alterado e simplificado (classes em cores foram criadas, outras classes já existentes no trabalho base foram omitidas por simplicidade).

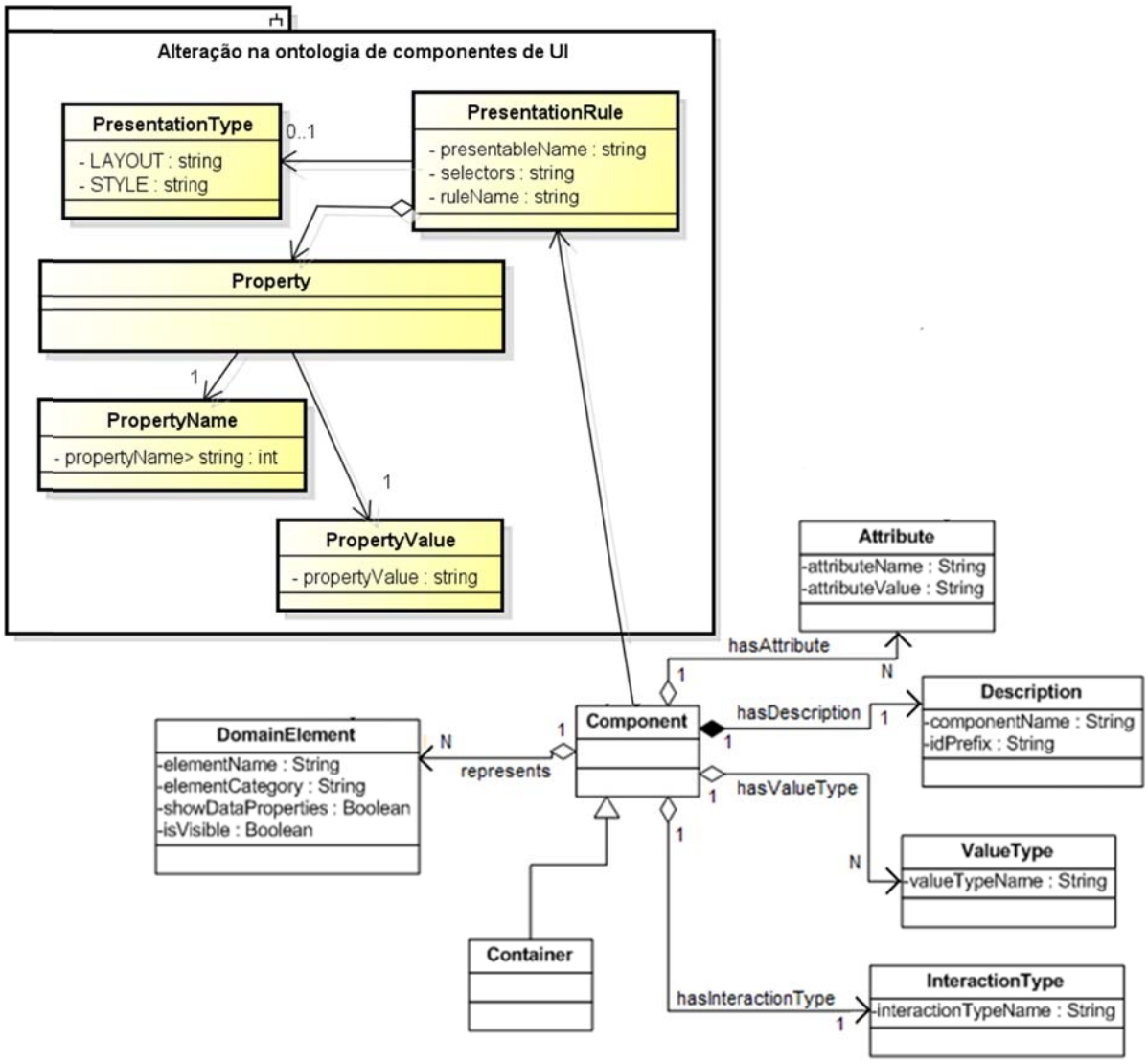

Figura 23 - Modelo ontológico de componentes de UI adaptado do MedViewGen (DUARTE, 2011)

A classe original *Component* representa um componente genérico de UI com seus atributos. A classe *Container*, especialização de *Component*, representa um componente agrupador, que pode conter componentes-filhos (DUARTE, 2011). O novo modelo deste estudo incorporou os itens comentados nas próximas seções. As classes seguintes estão incluídas também no diagrama da Figura 11, página 50.

3.3.1 Classe PresentationRule

A classe *PresentationRule* (regra de apresentação) (Figura 24) é composta de um conjunto de propriedades genéricas responsáveis pela apresentação visual da UI.

Tais propriedades são do tipo chave = valor, atribuíveis ao documento ou a cada região específica desse, mapeáveis para tecnologia concreta de folhas de estilo (CSS) usada em interfaces *web, que define layout e estilo.* O mapeamento está descrito adiante na seção 3.4

O dataproperty *presentableName* (string) é obrigatório e serve como identificador da regra, facilitando, inclusive sua localização na ontologia. Para diferenciar a *PresentationRule* destinada a definir isoladamente *layout* daquela para estilo, foi estabelecido o relacionamento opcional *hasPresentationType* com indivíduos da classe *PresentationType* (do tipo Layout ou do tipo Style), considerando a recomendação estabelecida para separação de *layout* e estilo.

As alterações propostas no modelo de UI também levaram em consideração a tecnologia ou especificação final a ser usada. Por isso, parte das definições de *layout*/estilo foi baseada em CSS, usada em sua grande maioria para estilo de interfaces web. A remodelagem realizada permite configurações diretas da tecnologia no modelo ontológico, sem que, no entanto, este deixe de ser genérico.

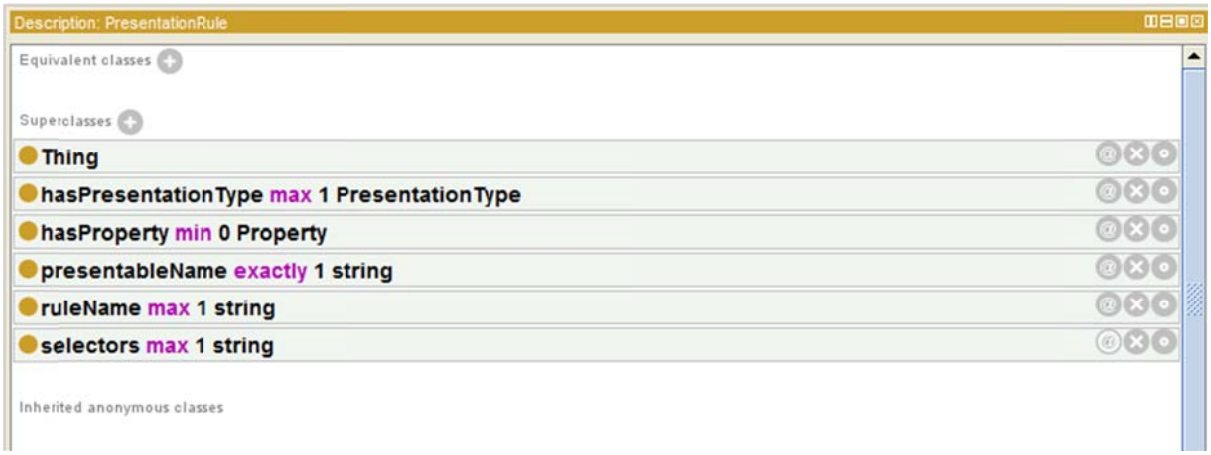

Figura 24 – Definição da classe PresentationRule

Considerando que no caso de CSS existem seletores de atributos, de elementos, de classes, de identificadores de elementos HTML ou mesmo o seletor universal, à classe *PresentationRule* foi atribuído o *dataproperty* opcional *selectors* (*string*), que permite que o *designer* insira seletores com a sintaxe da linguagem da tecnologia final de *layout*/estilo. Essa técnica é conhecida como *pass through*, porque não há nenhuma tradução, sendo a propriedade atribuída à tecnologia concreta do mesmo modo como foi inserida genericamente. O *pass through* é usado

em JSF, no atributo *style* de objetos da classe *javax.faces.UIComponent*, de modo que o programador deve escrever com a mesma sintaxe CSS os atributos com seus valores em uma única linha, considerando o uso do renderizador original do JSF para HTML. Essa técnica evita a necessidade de memorização adicional de uma sintaxe genérica que, no final, seria traduzida de todo modo para CSS, que é a linguagem de *layout*/estilo nativa dos navegadores *web*.

O atributo opcional *ruleName* (*string*) serve como nome da regra, podendo ser usado para identificar ou classificar um conjunto de propriedades de *layout*/estilo. Se não houver *seletors*, o algoritmo usa o valor de ruleName como nome da classe CSS para classificar as propriedades contidas na regra. Se não houver ruleName, o algoritmo insere os atributos através de estilo *inline* quando da conversão para tecnologia de interface gráfica e de *layout*/estilo concreta. O estilo inline é um recurso em que todas as propriedades são serializadas em uma única *string* e inseridas diretamente no elemento de UI, como atributo. A especificação CSS é rica e flexível, cabendo a ela toda uma descrição ontológica à parte bastante complexa.

Contudo, para microestruturas de UI, visando não restringir em absoluto a modelagem, consideramos permitir, na prática, a *PresentationRule* sem tipo, sem o relacionamento *hasPresentationType,* para permitir definições rápidas de adornos *ad hoc* simples para algum componentes de UI que não chegam a interferir na aparência do documento como um todo.

3.3.2 Classe Property

Cada *PresentationRule* possui ao menos uma *Property* (Figura 25) que, por sua vez, foi modelada de modo a evitar inserções repetidas de propriedades chave=valor, poupando tempo e minimizando a ocorrência de possíveis e recorrentes erros de digitação. O relacionamento *hasPropertyName* com *PropertyName* modelada como classe em vez de atributo (*dataproperty*) permite que *PropertyName* tenha seus indivíduos inseridos manualmente uma única vez no *pool*, sendo reutilizáveis em diversos indivíduos *PresentationRule*. A classe *PropertyName* (Figura 26) tem apenas um *dataproperty*, *propertyName* (string), obrigatório, para o nome da propriedade genérica que se deseja atribuir.

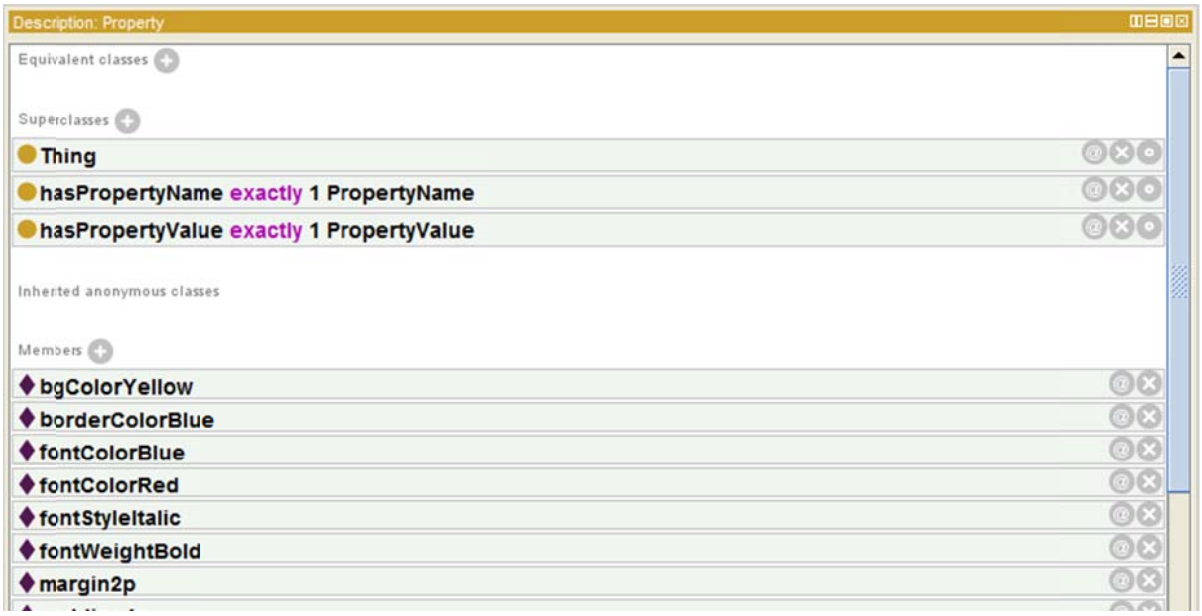

Figura 25 – Definição da classe Property

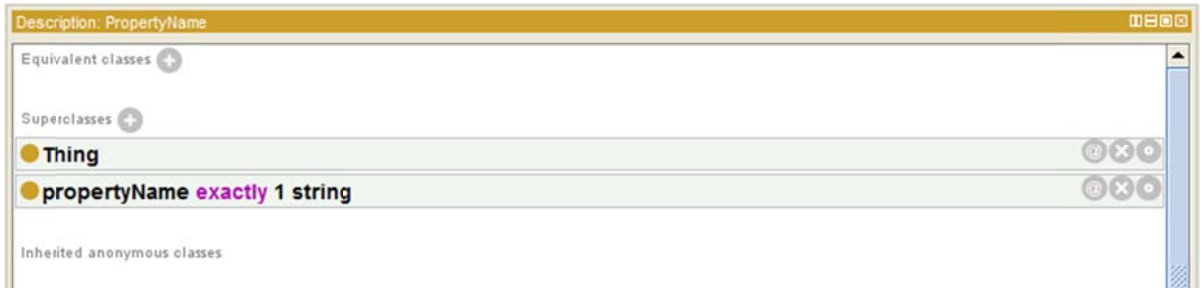

Figura 26 – Definição da classe PropertyName

Abordagem semelhante foi aplicada a *PropertyValue* (Figura 27), que tem um só *dataproperty*, *propertyValue* (string), obrigatório, que recebe o valor inserido que também vai para o *pool* de indivíduos, sendo reutilizáveis. Ressalta-se que o valor inserido juntamente com a sua unidade é atribuído em tempo de execução pelo algoritmo AAC via *pass through*, mencionado anteriormente.

Figura 27 – Definição da classe PropertyValue

Um exemplo do uso de *Property* e suas dependências está ilustrado no Quadro 2.

Nome do indivíduo	**ObjectProperty**	**Nome do indivíduo (Classe)**	**dataproperty**
fontWeightBold	hasPropertyName	fontWeight (PropertyName)	propertyName = “fontWeight”
	hasPropertyValue	_bold (PropertyValue)	propertyValue = “bold”

Quadro 2 - Um exemplo do uso de Property e suas dependências

O indivíduo *fontWeightBold***,** por exemplo, pode ser reutilizado em qualquer outra *PresentationRule*, pois é uma propriedade completa, com chave genérica e valor real (atribuível via *pass through*) .

Em termos de UI genérica, a árvore de componentes autogerada como objetos Java recebe, em sua raiz, um objeto com todas as propriedades genéricas de *layout* e estilo, que serão aplicados somente na etapa posterior de conversão de UI genérica para UI concreta, no caso, para a tecnologia JSF. A UI genérica é apenas uma especificação, não sendo renderizável em nenhum navegador ou visualizador.

3.4 Modelo ontológico para mapeamento de tecnologias de UI

O modelo ontológico da Figura 28, implementado no OpenCTI para mapeamento de tecnologias de UI, se relaciona ao modelo componentes da Figura 23 (página 65) através da classe *GenericComponent*, que representa um componente de UI, tendo sido mapeados para tecnologia de componentes JSF no trabalho original. Este modelo, entretanto, foi alterado para o presente estudo de modo a incluir o mapeamento de propriedades genéricas de *layout* e estilo para CSS que é a especificação dominante para interfaces web.

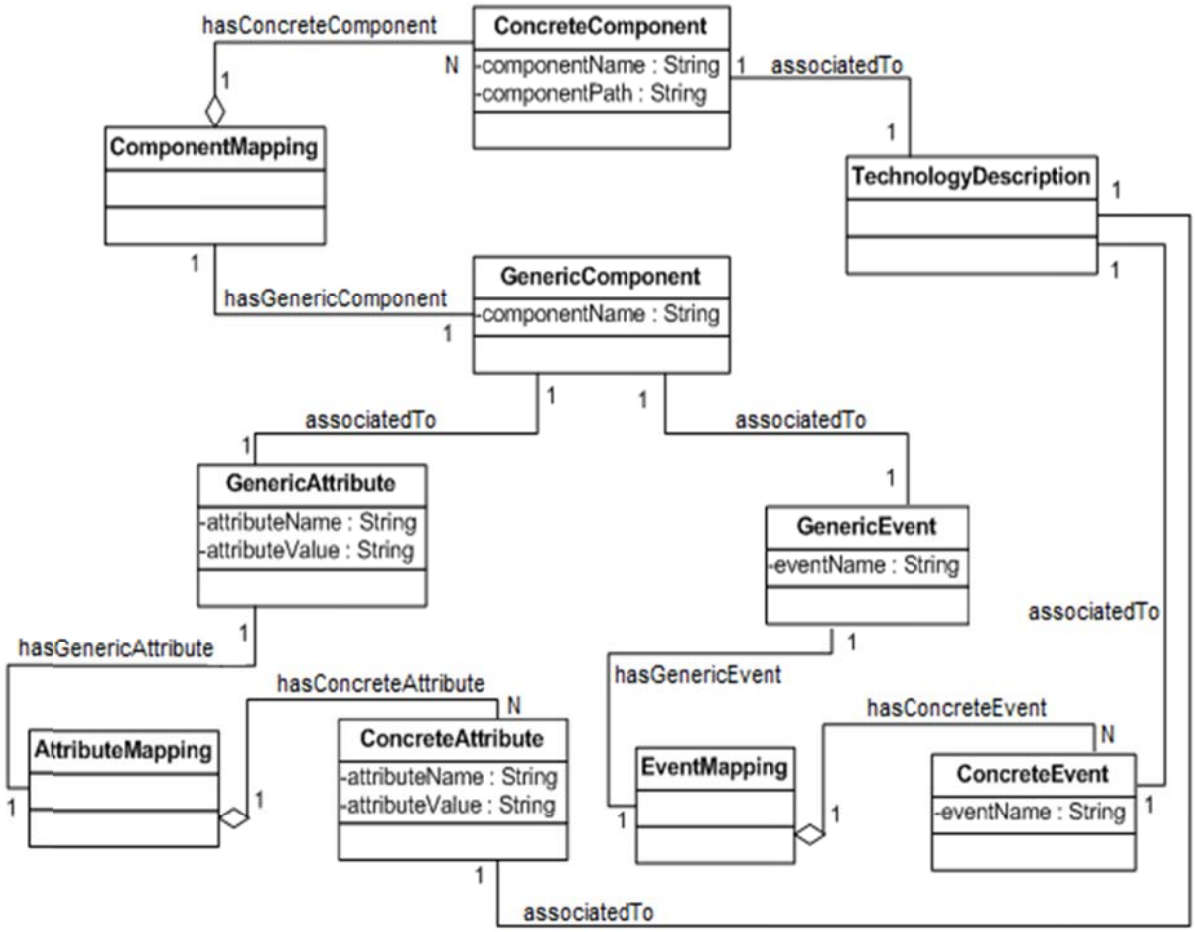

Figura 28 - Modelo de mapeamento de tecnologia de componentes de UI (DUARTE, 2011)

Considerando que a classe *ConcreteAtribute* do modelo original possui uma aplicação diferente (para atributos de componentes concretos), optamos por adicionar uma outra para identificar nome de propriedade concreta, *ConcretePropertyName*, conforme ilustrado na Figura 29:

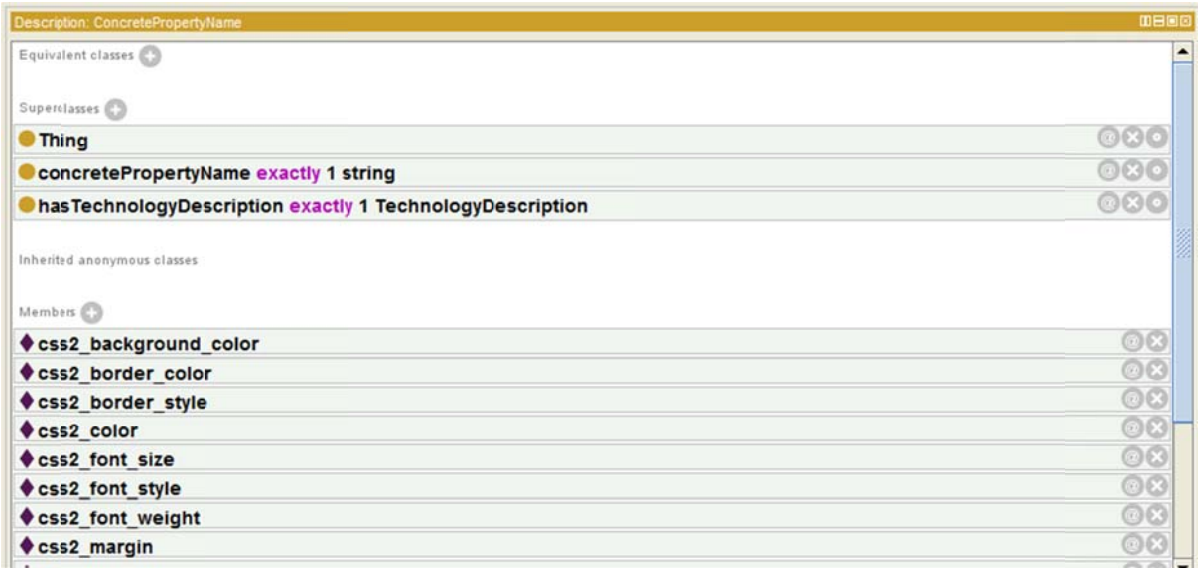

Figura 29 – Definição da classe ConcretePropertyName

O *dataProperty concretePropertyName* (string) é obrigatório e corresponde ao nome de uma propriedade CSS. O atributo obrigatório *hasTechnologyDescription* (string) corresponde ao nome/versão da tecnologia/especificação correspondente à propriedade, neste caso, atribuímos CSSx, que corresponde às versões 2 e 3.

O mapeamento entre propriedade de *layout*/estilo genérica e concreta é estabelecido na classe *PropertyMapping* (Figura 30):

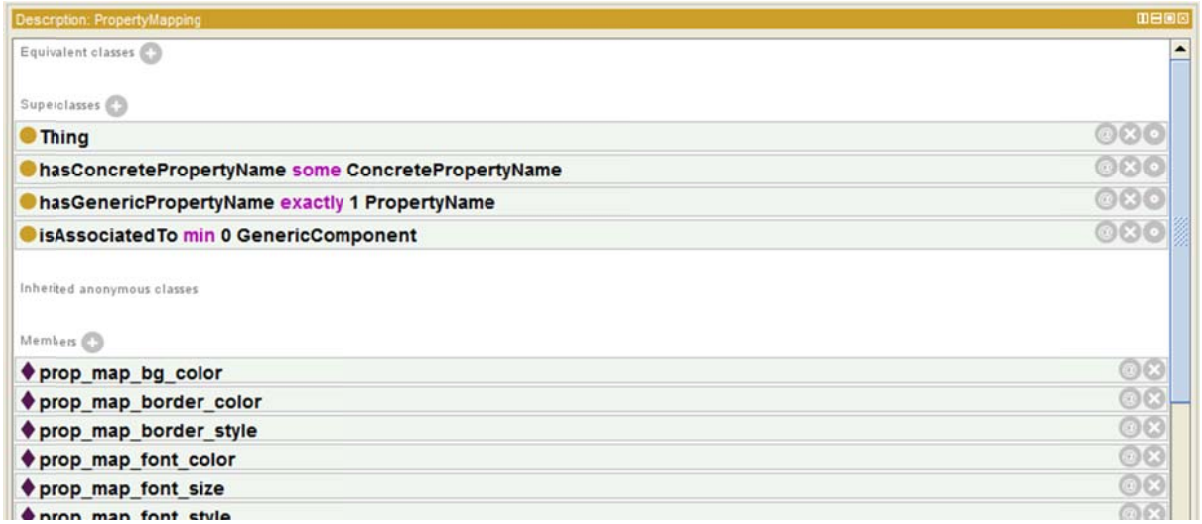

Figura 30 – Definição da classe PropertyMapping

O relacionamento *hasGenericPropertyName* estabelece a ligação entre *PropertyName* (genérico) para o mapeamento com *ConcretePropertyName* (string)

que corresponde ao nome da propriedade concreta CSS. Como exemplos de mapeamento, o Quadro 3 fornece alguns exemplos:

Nome do indivíduo na ontologia	**Nome genérico da propriedade**	**Nome da propriedade em CSS**	**Representa**
prop_map_bg_color	bgColor	background-color	Cor do plano de fundo
prop_map_font_size	fontSize	font-size	Tamanho da fonte
prop_map_font_weight	fontWeight	font-weight	Espessura da fonte
prop_map_font_color	fontColor	Color	Cor da fonte

Quadro 3 - Exemplo de mapeamento entre propriedades genéricas e concretas de *layout*/estilo CSS

No Apêndice I (página 109), foi descrito um pequeno dicionário com mapeamento das propriedades usadas neste trabalho e outras observações. Não foram incluídos nomes concretos de propriedades específicas de cada fornecedor/fabricante de navegador web, como aquelas com prefixo -webkit- para Safari (renderizador *WebKit*) ou –moz- para Mozilla Firefox (renderizador Gecko), já que seria assunto para uma modelagem adicional.

Em termos de processamento em *software, e*ssas propriedades CSS são atribuídas no momento posterior à tradução da árvore genérica de componentes para JSF. São atribuídas como estilo *inline* para cada componente ou documento, quando especificado, ou como estilo definido por classes. Foi necessária a codificação de um conversor para CSS para geração automática de um arquivo de folha de estilo externa, cujo conteúdo representa as classes em bloco, com suas propriedades e valores na sintaxe da especificação. A carga deste arquivo na página *web* foi realizada através de requisição assíncrona (AJAX). A Figura 33, na página 77, exibe as mensagens para chamadas de métodos do componente *CSSConverter*.

3.5 Considerações finais

Nas subseções anteriores, foram descritas as contribuições realizadas no presente estudo tendo como base os modelos ontológicos desenvolvidos por Duarte (DUARTE, 2011) e Nóbrega (NÓBREGA, 2010). O Quadro 3 resume as principais alterações realizadas em cada ontologia e algumas observações:

Ontologia	**Trabalho/autor original**	**Alterações realizadas no presente estudo**
Modelo de conceitos biomédicos (Figura 9)	(NÓBREGA, 2010)	Inserção de atributos para valor máximo arbitrário de conceitos biomédicos quantitativos para determinar tamanho máximo de componentes de entrada de dados.
Modelo de documentos (Figura 11)	(DUARTE, 2011)	Criação de novas classes e indivíduos para modelagem de formatos de apresentação para área médica (*Presentation*, *PresentationFormat*, *ConceptDisplayUnit*, *DocumentSymbol*, *GridLayout*, *PresentationSet*) e adaptações necessárias em outras classes existentes.
Modelo de componentes de UI (Figura 23)	(DUARTE, 2011)	Criação de novas classes e indivíduos para modelagem de formatação de componentes de UI com *layout* e estilo (*PresentationRule*, *Property*, *PropertyName*, *PropertyValue*, *PresentationType*). e adaptações necessárias em outras classes existentes.
Modelo de mapeamento para tecnologias de UI (Figura 28)	(DUARTE, 2011)	Criação de novas classes e indivíduos para mapeamento para a especificação CSS usada no presente trabalho (*ConcretePropertyName*, *PropertyMapping*).

Quadro 4 - Resumo das contribuições do presente estudo para modelagem ontológica

4

Arquitetura e implementação

Neste capítulo, apresentamos e discutimos na seção 4 as alterações para adaptar a arquitetura do *framework* MedViewGen (DUARTE, 2011) a fim de permitir o uso das especificações de *layout* baseadas nos modelos ontológicos estabelecidos no capítulo anterior. Em seguida, na seção 4.2, apresentamos e discutimos as alterações no algoritmo de atribuição de componentes (AAC) do trabalho original (DUARTE, 2011) para tornar possível a atribuição de componentes com uso de indivíduos das novas classes incluídas nos modelos ontológicos, bem como atribuições de restrições e propriedades para *layout* e estilo.

4.1 Arquitetura do sistema

Antes de determinar uma arquitetura para o gerador de *layout*/estilo, optamos por avaliar a arquitetura já existente para o gerador de UI do projeto OpenCTI, o MedViewGen (Figura 31), desenvolvido por (DUARTE, 2011). Parte dela foi descrita na seção 2.3.5.1, página 36, para o OpenCTI como um todo.

No MedViewGen, podemos dividir o fluxo de execução em duas fases. A primeira é onde ocorre a construção da árvore de UI genérica. A geração da interface começa a partir de uma requisição de um documento pelo usuário (1). A requisição originada na UI (componente View) é encaminhada ao componente gerenciador da UI (GUIManager) (3) que é responsável por manter o estado atual dos documentos e informações do contexto do usuário, recebendo eventos da UI e repassando-os aos componentes responsáveis por seu processamento. O gerenciador de documentos (DocumentManager) (2) recupera metadados do documento em requisição, busca informações dos componentes genéricos de UI

armazenadas na ontologia e informações relativas aos elementos de CDS implementados. O componente de geração automática (GenericGUIBuilder), então, elabora um formato de interface genérico para a apresentação do documento (4).

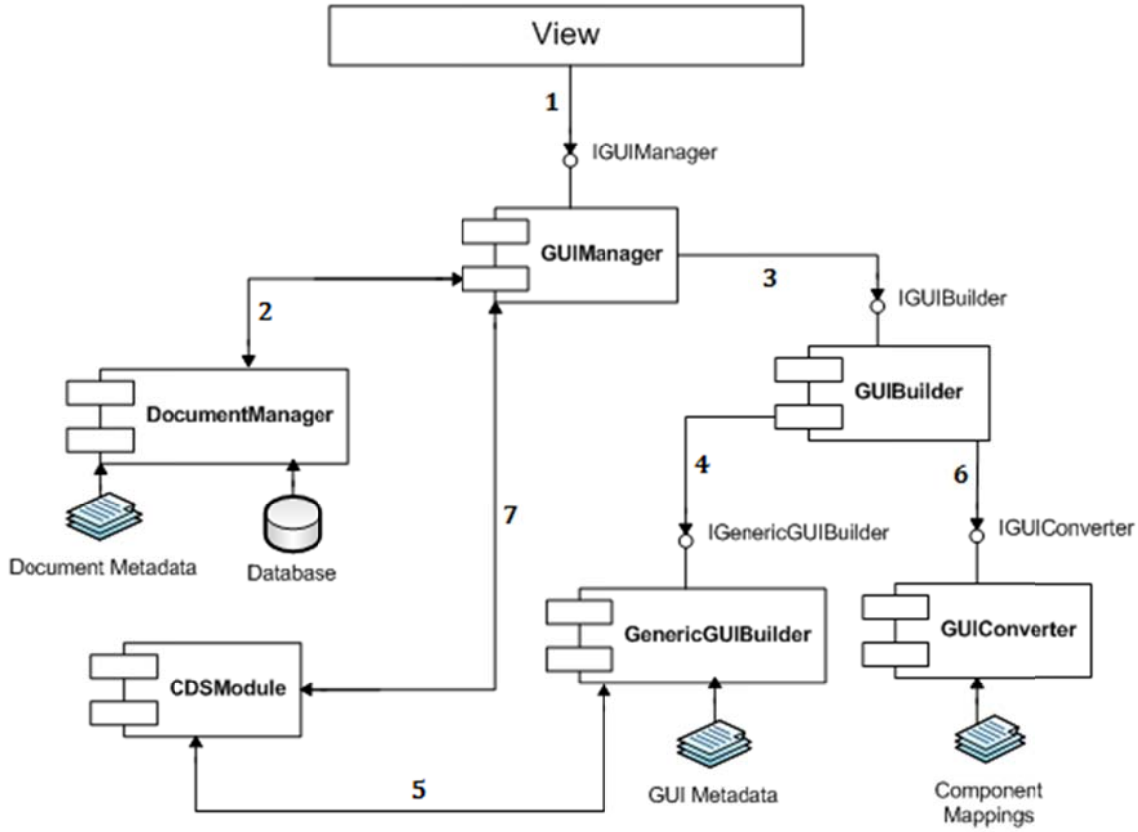

Figura 31 - Modelo de componentes do MedViewGen (DUARTE, 2011)

A fase 2 é onde ocorre a conversão da interface genérica para interface de alguma tecnologia específica de componentes. A estrutura da interface genérica completa é repassada para o GUIConverter (6) que traduz os componentes genéricos para componentes da tecnologia, com base nas informações da ontologia de mapeamento. Os componentes concretos são utilizados na etapa seguinte para construção do formulário final (GUIBuilder), que é enviado ao usuário como resposta à requisição, devidamente integrado à página *web* de onde partiu a requisição inicial ao documento (DUARTE, 2011).

Considerando a modularização arquitetural pré-existente, inserimos as restrições para o *layout*/estilo nos modelos ontológicos, pois algumas devem provir de definições de conceitos biomédicos, outras devem ser determinadas como

propriedades genéricas para o documento e/ou para cada item do documento. Optamos, portanto, realizar a extração das restrições para o *layout*/estilo durante a execução do algoritmo AAC, tendo sido necessário, para isto, criar alguns componentes, recodificar alguns métodos já existentes e implementar outros.

A Figura 32 ilustra de forma alternativa o processo para geração de UI genérica após uma requisição de um documento pelo usuário, através de um diagrama de sequência em UML (itens em magenta foram criados e/ou alterados para o presente trabalho).

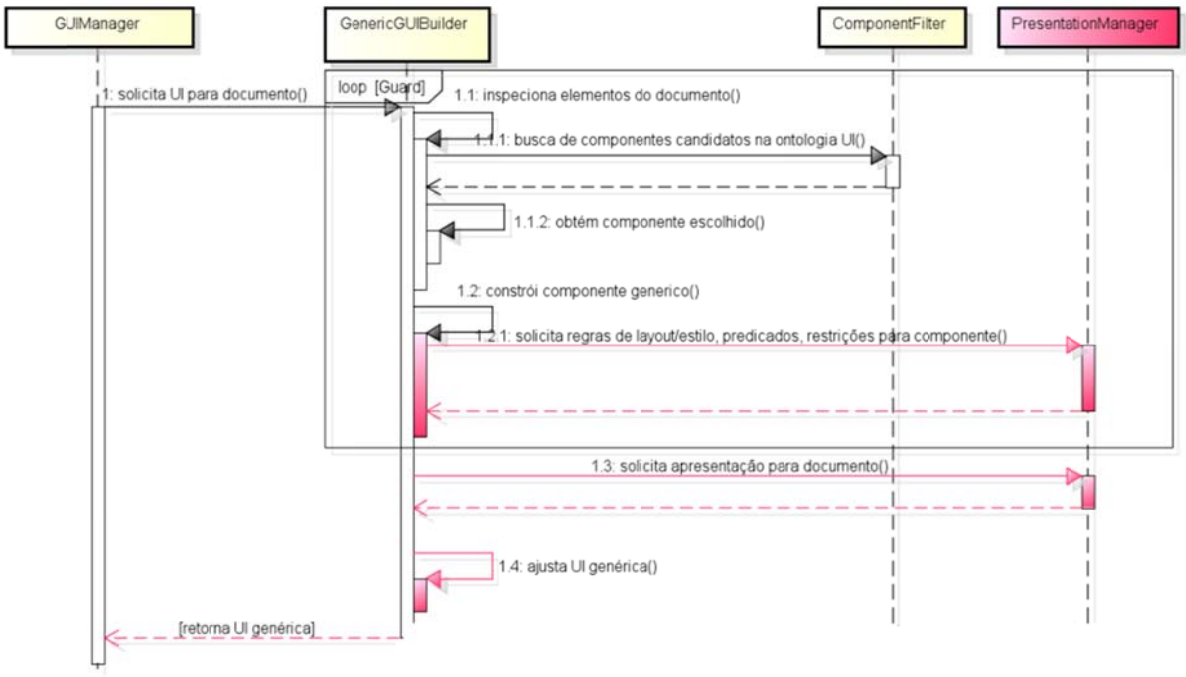

Figura 32 - Diagrama de sequência para geração de UI genérica com *layout*

Após a construção da UI genérica, o sistema solicita a conversão em UI concreta para JSF, através de um componente já existente chamado *JSFBuilder*, conforme diagrama de sequência UML da Figura 33. Os itens em magenta foram criados ou alterados para comportar a geração do *layout*/estilo em especificação concreta CSS.

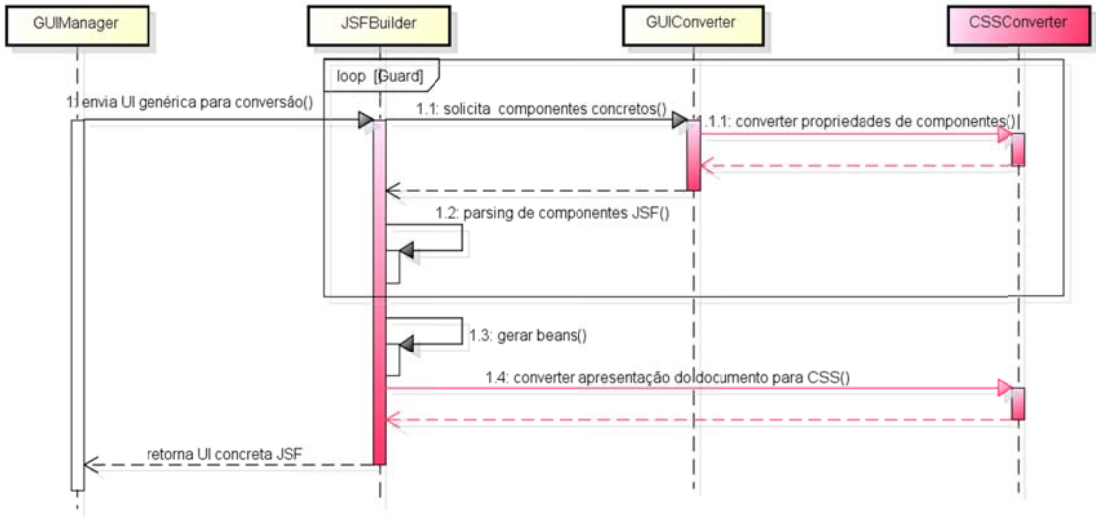

Figura 33 - Diagrama de sequência para geração de UI concreta com *layout*

Alguns componentes adicionais foram omitidos das figuras anteriores por simplicidade, já que não se fazem necessários para compreensão das sequências.

Em resumo, as contribuições do presente estudo ao trabalho original tomado como base foram a criação e codificação de novos componentes de *software* para funcionamento em conjunto com aqueles existentes na arquitetura original, conforme explicado nas seções seguintes.

4.2 Algoritmo de atribuição automática de componentes

Discutiremos nesta seção algumas alterações importantes realizadas no presente estudo no algoritmo de atribuição de componentes do trabalho original (DUARTE, 2011) para tornar possível a atribuição de componentes com uso de indivíduos das novas classes incluídas nos modelos, bem como atribuições de restrições e propriedades para *layout* e estilo.

Os métodos mencionados a seguir referem-se a filtragens realizadas seguindo diversos critérios no trabalho original. Um conjunto de componentes genéricos é carregado da ontologia e, de acordo com os elementos presentes no documento solicitado, são selecionados componentes de interface candidatos para cada elemento, até que reste apenas um, que será incluído na estrutura da UI.

4.2.1 Filtragem por classe

O algoritmo original da filtragem pela classe não foi alterado, mas a inserção de novos itens nas ontologias permitiu a atribuição de novos componentes. Para isto, foi necessário incluir na ontologia de Componentes as classes *PresentationFormat*, *ConceptDisplayUnit*, *DocumentSymbol* como *DomainElement*, de modo que tenham componente genérico de UI que possam representá-las em um documento.

Para a classe *PresentationFormat*, definimos um componente do tipo *panel* para representá-lo. O *panel*, na ontologia de componentes, é um contêiner de mais alto nível, servindo também para representar itens das classes *Header* (cabeçalho do documento), *AbstractBiomedicalConcept* (conceito biomédico abstrato), além da classe *Archetype* (arquétipo) na versão original do trabalho.

A classe *ConceptDisplayUnit* foi referenciada na ontologia de componentes de UI como um indivíduo da classe *DomainElement* (elemento de domínio presente no documento), e, portanto, tem um componente genérico (classe *Component*) que pode representá-lo em uma UI genérica. No estudo em questão, um tipo de componente genérico chamado *box* foi criado, que é um contêiner sem cabeçalho, não necessariamente visível, mapeado para JSF como *javax.faces.component.html.HtmlPanelGroup* com atributo concreto *layout* = “*block*”, que permite que o JSF renderize um elemento HTML. Esta informação é utilizada pelo algoritmo de AAC.

Outros componentes foram criados para comportar a exibição de DocumentSymbol, conforme mencionado na seção 3.2.3.

Portanto, as alterações já descritas fornecem informações ao algoritmo AAC para realizar atribuição de componentes genéricos novos para suportar as modificações realizadas em todo o modelo. Nas seções seguintes, são mencionadas novas alterações relacionadas.

4.2.2 Filtragem por atributo

Houve necessidade de incluir um novo filtro para dar suporte à atribuição de componente para elemento do tipo *DocumentSymbol*. De acordo com o atributo presente, um componente diferente é escolhido.

Para representar um *DocumentSymbol*, conforme descrito previamente, temos um componente genérico chamado *image* que comporta uma URL e exibe um arquivo de imagem e um componente de exibição de texto, do tipo *outputText*, para caracteres textuais.

Em termos de tecnologia JSF, o componente genérico *image* é mapeado para o componente concreto representado por objeto da classe *javax.faces.component.html.HtmlGraphicImage*, que renderiza um elemento HTML <img>, para imagens, e *outputText*, para *javax.faces.component.html.HtmlOutputText*, que renderiza um elemento HTML , apropriado para texto. Por isto, a necessidade deste filtro para comportar esta diferenciação. A Figura 34 exibe parcialmente a marcação HTML final gerada pelo JSF de acordo com a modelagem proposta para o formato de apresentação do conceito da pressão arterial. As setas na figura indicam os elementos HTML mencionados (<img> e).

```
1  <div id="new_doc_include:PA4_header" class="ui-panel-titlebar ui-widget-header ui-corner-all ui-helper-clearfix">
2  <span class="ui-panel-title">Pressão arterial</span></div>
3
4  <div id="new_doc_include:PA4_content" class="ui-panel-content ui-widget-content">
5  <table id="new_doc_include:PG32" cellspacing="10"><tbody><tr>
6
7      <td><div id="new_doc_include:BOX7">
8
9
10     <img id="new_doc_include:IMAGE8" src="styles/default/img/sphygmomanometerLabel.png"></div></td>
11
12     <td><div id="new_doc_include:BOX9">
13         <table id="new_doc_include:PG33" cellspacing="10"><tbody><tr>
14
15             <td><div id="new_doc_include:BOX14" style="font-size: 1.5em;">
16                 <input id="new_doc_include:IT15" name="new_doc_include:IT15" size="3" style="font-size: 1.5em;" type
17
18             <td><div id="new_doc_include:BOX12" style="font-size: 1.5em;">
19
20
21                 <span id="new_doc_include:OT13" style="font-size: 1.5em;">/</span></div></td>
22
23             <td><div id="new_doc_include:BOX10" style="font-size: 1.5em;">
24                 <input id="new_doc_include:IT11" name="new_doc_include:IT11"  size="3" style="font-size: 1.5em;"
25
```

Figura 34 - Marcação HTML (parcial) gerada para o formato de apresentação da pressão arterial

4.2.3 Filtragem pelo tipo de interação e de valor

A filtragem pelo tipo de interação[4] foi alterada para que pudesse atribuir componente do tipo caixa de texto para elementos com tipo de valor *string* com tamanho maior que 100 (arbitrário).

Para isto, foi inserido na ontologia de conceitos biomédicos um *dataProperty*, *maxValue*, para conceitos qualitativos numéricos, de modo que fosse possível calcular o valor máximo do conceito em comprimento de caracteres. Para conceitos qualitativos do tipo *string*, foi atribuído um dataProperty, *maxLength,* para o valor máximo de caracteres permitidos, o que leva à atribuição automática de um componente genérico que chamamos *textbox* (caixa de texto), que pode ser traduzido para JSF/HTML como um inputTextarea/textarea, que corresponde a uma caixa que permite inserção de caracteres em múltiplas linhas. Um equivalente mais elaborado a este componente seria um editor de texto rico (*richText editor*), com suporte à formatação do conteúdo.

Para esta alteração, foi repetida a filtragem usando outro argumento para o que foi chamado de tipo de interação (DUARTE, 2011), através do indivíduo ontológico *free_multi_line*, que corresponde a um valor em forma de texto livre com apresentação de caracteres em mais de uma linha, o que normalmente acontece com textos longos, como, por exemplo, campos para escrita de observações em geral, impressões clínicas, laudos, *etc*. Arbitrariamente, para efeitos do presente estudo, assumimos texto longo contendo mais de 100 (cem) caracteres.

Com isso, ficou sob a responsabilidade do *filtro por valor* do algoritmo AAC a escolha entre um componente do tipo *inputText* e do tipo *textbox*, em caso de mais de um candidato na etapa final da atribuição. Para resolver essa situação, foi alterado o referido filtro de modo a incluir como componente candidato o primeiro componente, *inputText*, para valores numéricos ou textuais com tamanho de até 100

[4] No trabalho original, foi atribuída a expressão *tipo de interação* ao tipo de dado que o componente de UI suporta, seja pra inserção ou exibição, conforme modelagem ontológica pré-existente: free (texto livre), many_of_a_list (escolha de mais de um item de uma lista de opções), one_of_a_list (escolha de apenas um item de uma lista de opções), output (dado de saída) etc. Entretanto, consideramos como tipo de interação o modo como o usuário interage com algum componente de UI, como pelo toque (touchscreens), comando por voz, inserção direta de dados, seja por digitação ou por dispositivo apontador (mouse). Mantivemos a nomenclatura original dos indivíduos, manifestando, no entanto, esta ressalva.

caracteres (definidos na ontologia ou arbitrariamente atribuído) e, acima disto, componente *textbox*. Usamos nesta etapa o tamanho máximo do dado para escolha de componentes específicos, mas não do seu tamanho na tela.

A Figura 38, página 87, mostra o formulário de exemplo com um *presentationFormat* para observações clínicas contendo uma caixa de texto.

Ressaltamos que, na prática, no caso de texto livre, convém não determinar o tamanho, pois o profissional é que sabe quanto deve escrever sobre seu paciente, deixando para a camada de segurança da aplicação tratar questões específicas, para evitar *buffer overflow*, etc.

4.2.4 Filtragem por valor

Componentes que comportam valores numéricos de conceitos são selecionados por este tipo de filtro. Uma pequena alteração foi incluída para busca da restrição de *layout*, que é o valor máximo (numérico) do conceito biomédico quantitativo relacionado ao componente, para impor um atributo para determinar o tamanho fixo do componente de entrada de dados. O valor inicial deste atributo é o comprimento do valor máximo convertido para *string*.

Neste caso, só componentes de tipo *inputText* são escolhidos, por serem adequados à inserção de valores pelo usuário.

Até aqui foram relatadas alterações necessárias no algoritmo AAC, para que novos componentes de UI pudessem representar os novos conceitos modelados. A seguir, será apresentado o modelo e a implementação para *layout* em *grid* dos documentos, com um algoritmo adicional.

4.3 *Layout* em *grid*

Para o presente trabalho, optamos por um sistema de grid simples para organização espacial dos componentes da árvore genérica de UI do documento. *Grid* significa grade e impõe uma restrição espacial aos componentes que organiza. Como um documento é dividido em seções, cada uma delas tem uma distribuição de conteúdo

que deve ser sempre a mesma para cada tipo de documento, em uma determinada plataforma, conforme questões de segurança da informação do paciente discutidas na seção 1.3.

Na ontologia de documentos, incluímos na classe Section, que representa a seção de um documento, um item adicional, conforme Figura 35, o relacionamento *hasGridLayout*, que atribui à seção um indivíduo do tipo *GridLayout*, que representa o grid para aquela seção, em que é definido apenas o número de colunas para o *grid*. Tem dois relacionamentos filhos:

- *hasLargeDeviceLayout*, para *layout* a ser exibido em dispositivos com alta resolução de tela;
- *hasSmallDeviceLayout*, para *layout* a ser exibido em dispositivos com resolução de tela pequena, pois é possível que alguns tipos de documento sejam exibidos smartphones ou tablets.

Portanto, de acordo com a resolução de tela do usuário, o algoritmo escolhe apenas um *layout*. Caso não tenha sido instanciado na ontologia, o sistema lança exceção. Essa classificação das resoluções de tela é arbitrária, apenas para finalidades de teste no presente estudo.

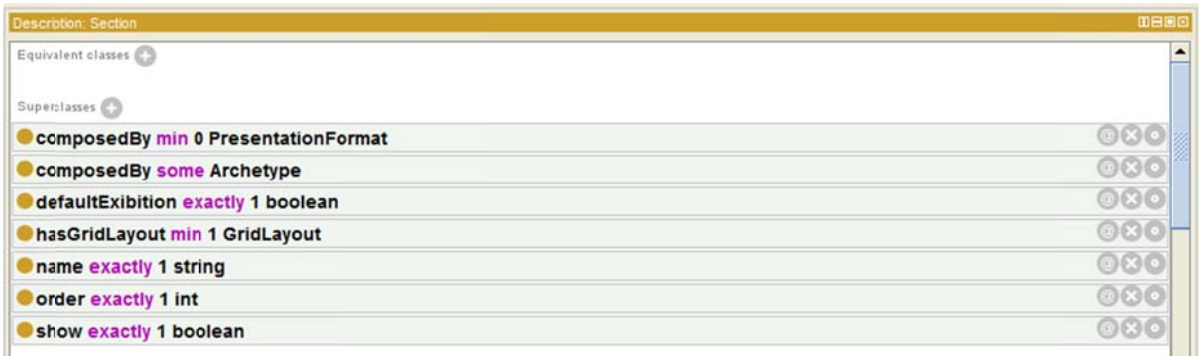

Figura 35 - Definição da classe Section

Optamos por aplicar o grid individualmente em cada seção porque cada uma delas pode ter itens (*PresentationFormat* e *PresentationSet*) em número e ordenação diferentes, de acordo com as necessidades dos usuários. Por isto, uma seção pode ter um número diferente de colunas, de acordo com *design* do

formulário. O *designer* deve ter em mente como quer que o formulário seja exibido, de acordo com o algoritmo de atribuição de conteúdo no *grid*.

4.3.1 Estrutura do *grid* e atribuição de conteúdo

O grid, com base no número de colunas definido na ontologia, é construído em tempo de execução. A Figura 36 mostra uma estrutura genérica dos itens inseridos dinamicamente pelo algoritmo de *layout* em *grid*.

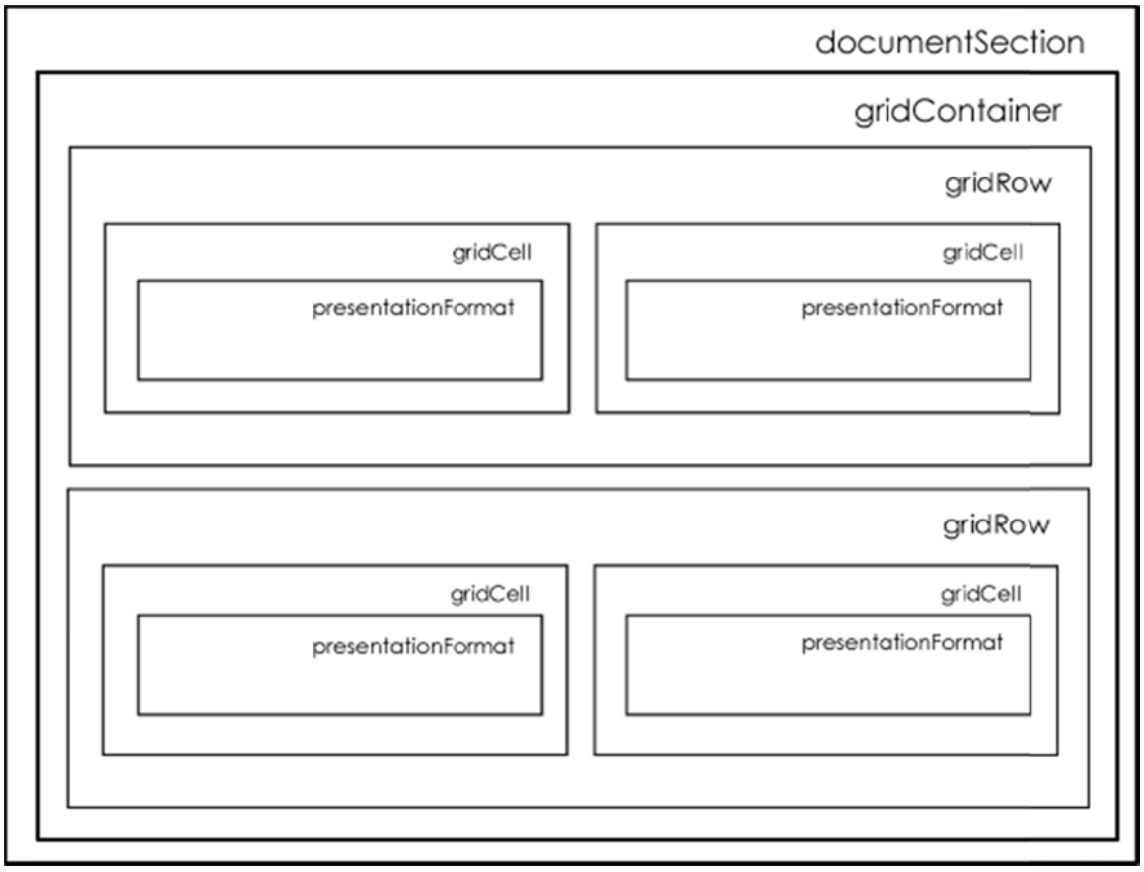

Figura 36 - Estrutura genérica do grid utilizado no *layout* dos documentos

Após a fase de atribuição de componentes, quando a UI genérica está definida, alguns ajustes são realizados, sendo a atribuição do grid de cada seção do documento o último deles. Os elementos estruturais (gridContainer, gridRow, gridCell) na figura anterior são adicionados, como objetos Java, à estrutura da árvore genérica de componentes de UI, de modo a agrupá-los dentro da seção do documento (*documentSection*) de acordo com o número de colunas definido na

ontologia e com a ordem de exibição dos arquétipos formatados (*PresentationFormat*). O item *gridContainer* representa o contêiner-raiz do grid dentro da seção do documento. Os itens *gridRow* representam as linhas horizontais do grid e contêm os *gridCell*, que representam as linhas verticais ou colunas de um *grid*. Dentro de cada *gridCell*, encontram-se os *presentationFormat* que compõem a seção de um documento. Optamos por não usar componentes do tipo tabela neste caso.

Opcionalmente, a classe *PresentationSet* quando presente na modelagem do documento define um conjunto de *PresentationFormat* que devem ser exibidos juntos em uma única linha.

No próximo capítulo são consolidados os resultados visuais do *layout* gerado para a interface gráfica, bem como discussão e comparação com outros trabalhos.

5

Resultados, discussão e comparação com outros trabalhos

Neste capítulo, na seção 5.1, apresentamos resultados visuais da aplicação automática de *layouts*, na seção 5.2 discutimos alguns pontos específicos do presente estudo e, finalmente, na seção 5.3, efetuamos uma breve comparação com outros trabalhos encontrados na literatura desenvolvidos com finalidade semelhante.

5.1 Resultados visuais

Os resultados visuais apresentados, em termos de especificação HTML/CSS, são renderizados como elementos HTML com propriedades CSS de *layout* pré-determinadas para que apresentem a estrutura do exemplo de documento da Figura 37.

A Figura 37 apresenta na primeira tela (metade superior da figura) um exemplo de documento com cinco elementos *PresentationFormat,* cujas linhas (*gridRow*) foram destacadas com borda vermelha e as células dentro das linhas (*gridCell*), com borda azul, para melhor visualização. Na segunda tela (metade inferior da figura), foi removido o destaque das linhas. Por fim, a Figura 38 apresenta o documento final sem realce de nenhum elemento do *grid*.

O formato de apresentação da última linha do documento da Figura 39 (observações clínicas) foi configurado na ontologia com uma propriedade booleana para alinhamento justificado, já que ficaria da largura das demais células, sozinho e alinhado à esquerda. Como contém uma caixa de texto para entrada de dados mais extensos, optou-se por determinar arbitrariamente que ocupe toda a linha em todos os casos. Esse alinhamento é conseguido através de propriedade CSS para largura

(*width*), atribuída via programação ao elemento da UI que representa a célula durante a construção do *layout*. Neste caso, o *grid* de duas colunas possui apenas uma na terceira linha, assemelhando-se a uma matriz esparsa e ainda à propriedade *colspan* de tabelas HTML.

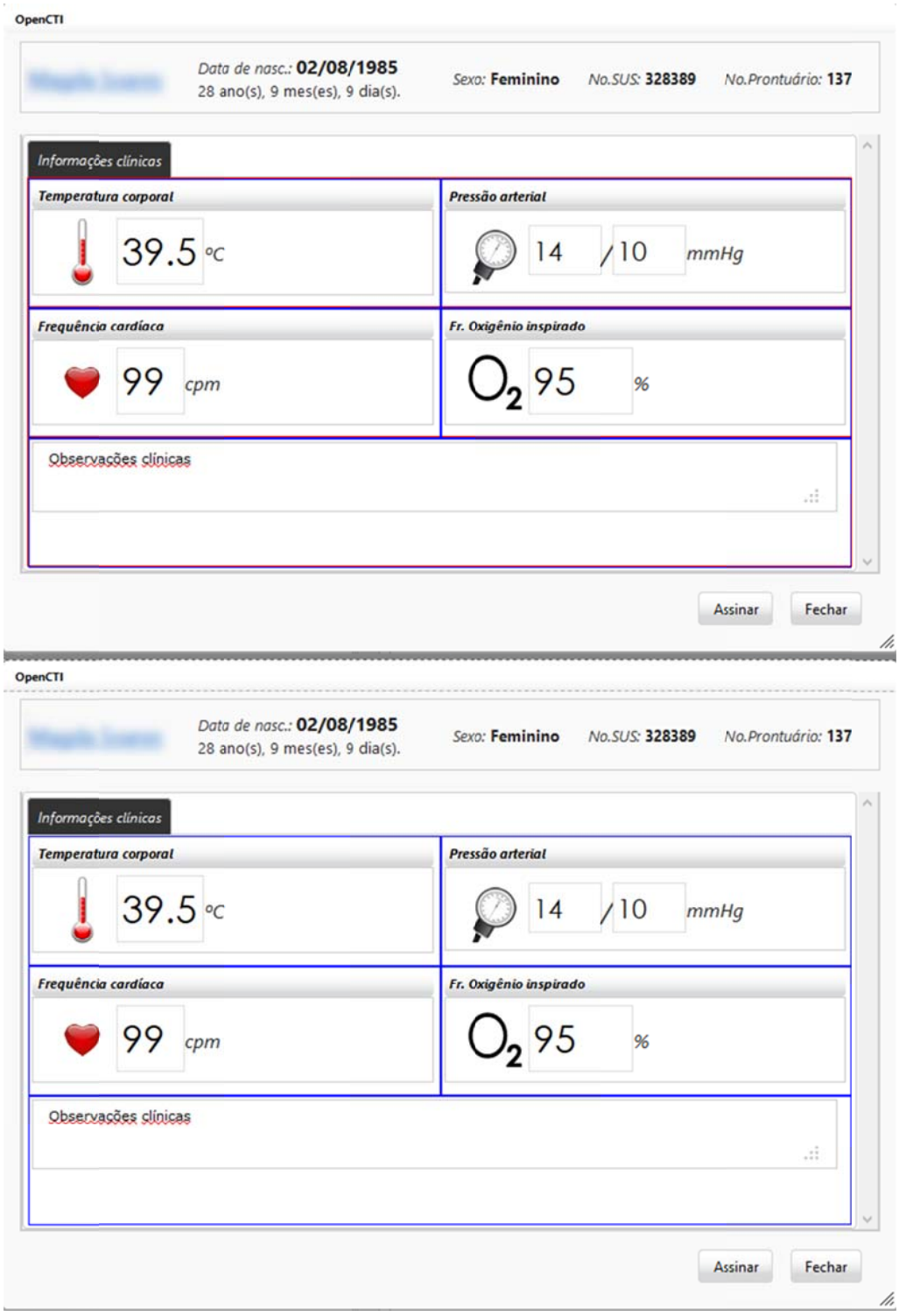

Figura 37 - Grid realçado em exemplo de documento

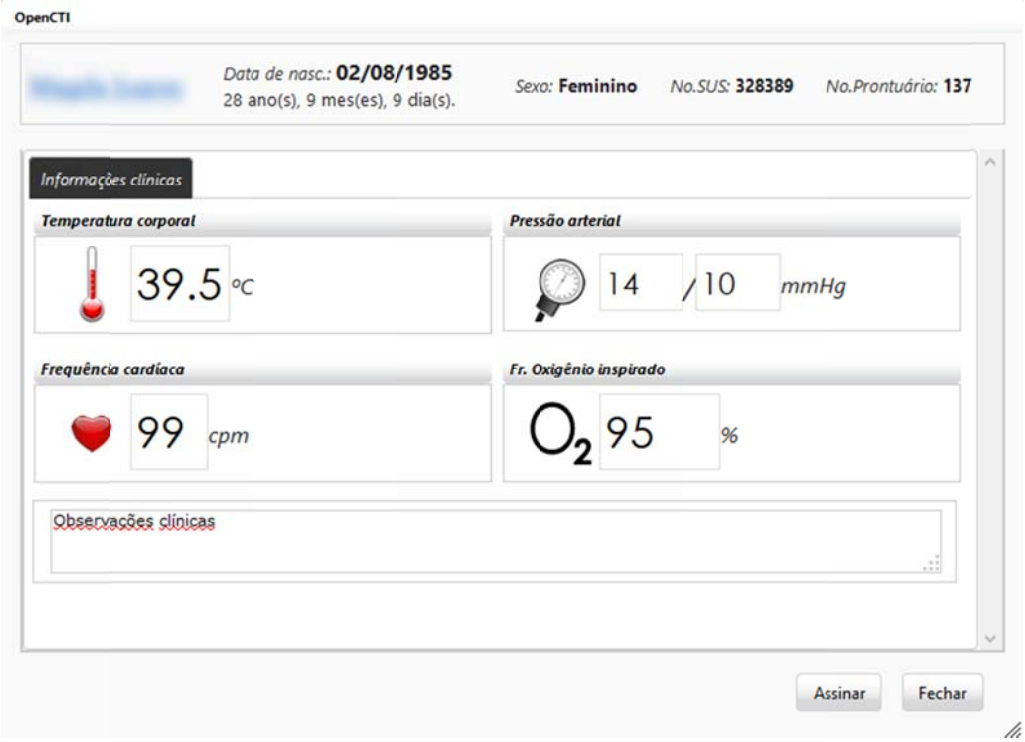

Figura 38 - Exemplo de formulário gerado dinamicamente com *layout* em grid

A seção do documento da Figura 39 também possui na ontologia um gridLayout definido para dispositivos pequenos, através do relacionamento *hasSmallDeviceLayout*, contendo apenas uma coluna, considerando o tamanho de tela menor desses dispositivos. Arbitrariamente, para finalidade de testes, atribuímos uma resolução de 580px e o sistema automaticamente escolheu o *layout* apropriado.

Como apenas uma coluna foi determinada para o *layout* do documento da Figura 39, o navegador exibiu a barra de rolagem para completa visualização do conteúdo. A razão para fixar o número exato de colunas é, além da limitação de resolução de tela, o fato de a rolagem horizontal ser bastante inconveniente e ainda um tanto incomum.

Por fim, a atribuição de conteúdo ao *grid* segue a ordem dos itens de cada seção definida para cada arquétipo ou formato de apresentação na ontologia, distribuindo os elementos já ordenados, da esquerda para a direita, obedecendo ao número de colunas também definido na ontologia. Verificamos que a ordem permaneceu a mesma nos dois exemplos de grid, com duas e com uma coluna (Figura 38 e Figura 39, respectivamente), para o mesmo documento, conforme inserido na ontologia.

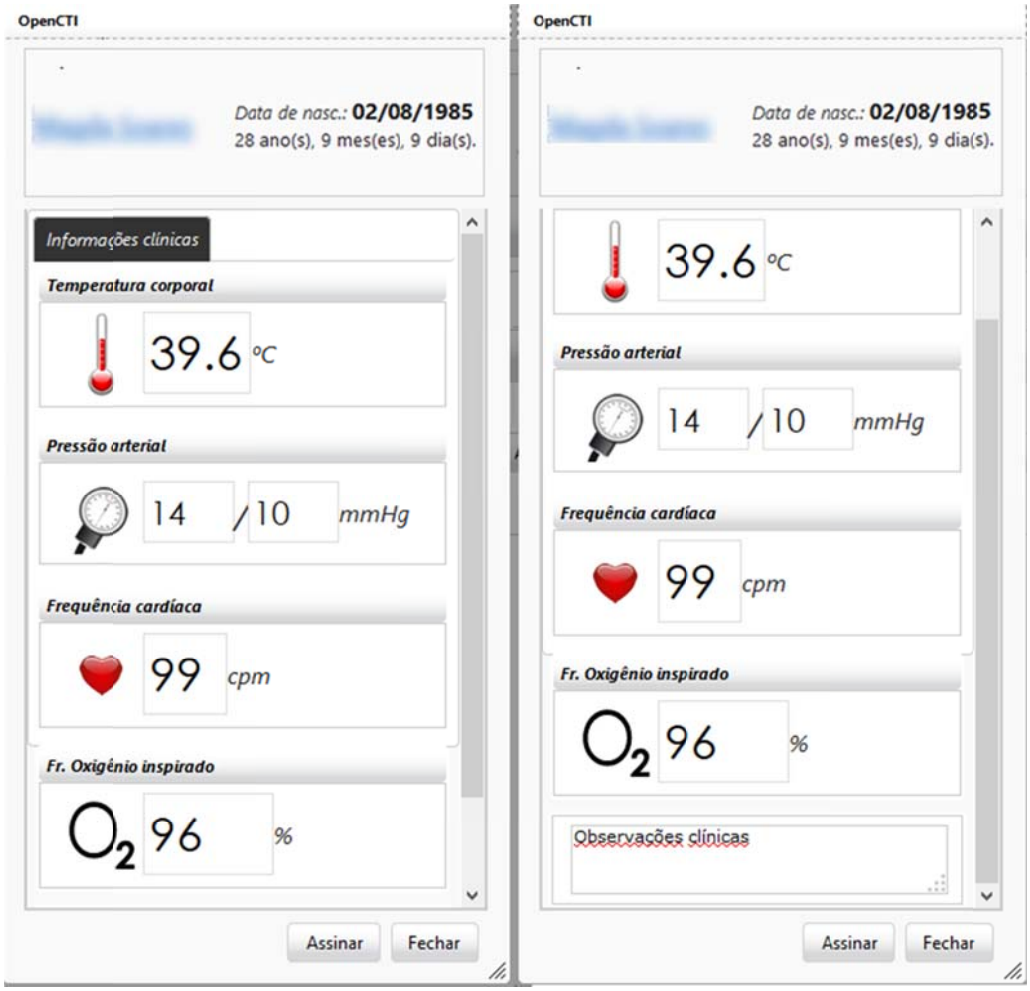

Figura 39 - *Layout* de documento para dispositivos de menor resolução

A Figura 40 ilustra através das capturas de telas parciais do software Protégé, a atribuição da propriedade de ordenação (*order*) de cada *PresentationFormat* do exemplo anterior de documento.

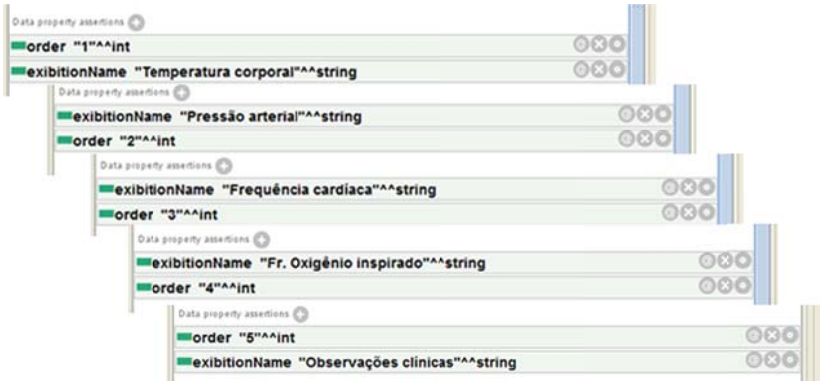

Figura 40 - Ordem dos formatos de apresentação na ontologia para o documento de exemplo

Ressalta-se aqui que uma parte do *layout* do documento é determinada pelo algoritmo de AAC original quando é escolhido para o mesmo o componente agrupador em abas para as seções do mesmo. Portanto, o grid do presente trabalho foi projetado para o conteúdo da seção do documento.

A Figura 41 mostra outro exemplo de formulário construído com uso de outras classes já definidas, como *PresentationSet*.

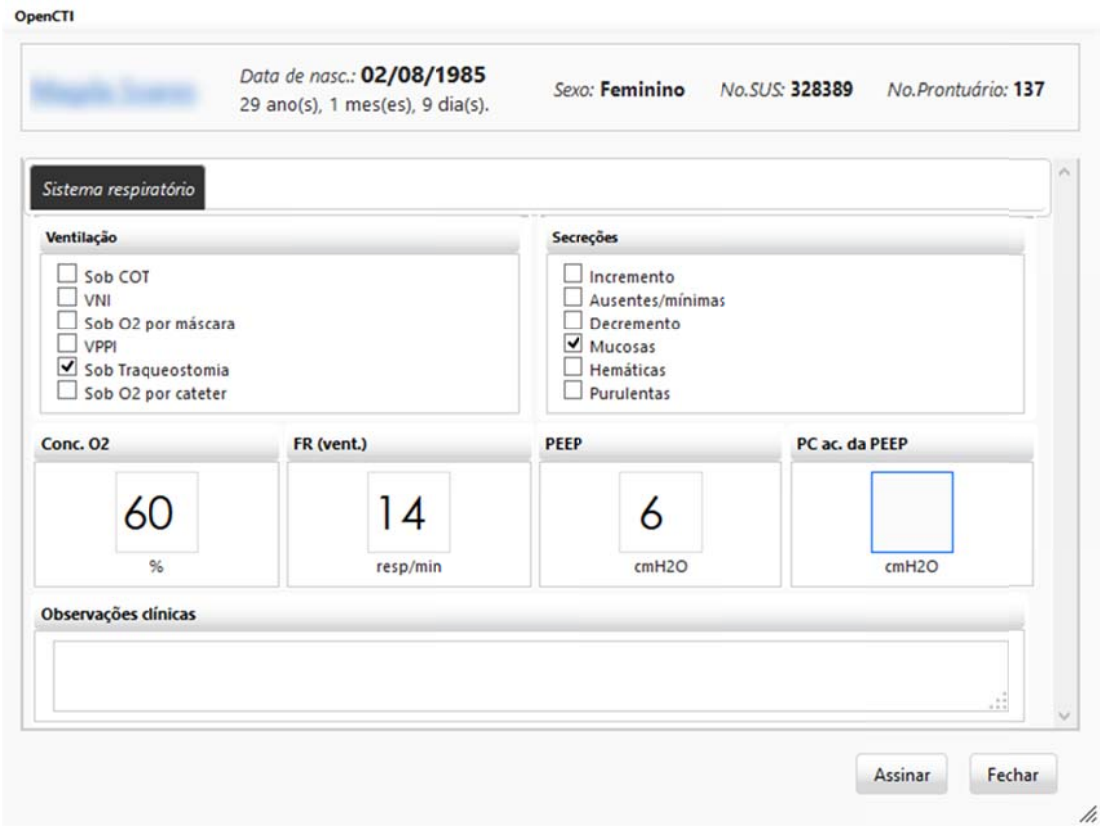

Figura 41 - Exemplo de formulário com uso de PresentationSet

Neste caso, atribuímos na ontologia duas colunas para o *grid* desta seção de documento. Foram inseridos na ontologia os dois primeiros itens como *PresentationFormat*, o seguinte como um indivíduo *PresentationSet* com order = 3 e quatro relacionamentos composedBy *PresentationFormat* para os parâmetros de ventilação para o paciente e, no final, mais um *PresentationFormat* para observações em texto livre. Portanto, a primeira linha apresenta dois itens (duas colunas conforme definição do grid desta seção na ontologia), a segunda, quatro por causa do *PresentationSet* e a terceira, apenas um. Neste último caso, foi determinado arbitrariamente que quaisquer conceitos representáveis por componentes genéricos do tipo *textbox* (caixa de texto) devem ser exibidos em uma

única linha, já que o conteúdo se trata de texto longo para o conceito biomédico observações clínicas.

Com relação ao conteúdo de cada componente *panel* (caixas com barra de título cinza) que representa os *PresentationFormat* da figura anterior, na primeira linha foram usados conceitos biomédicos compostos para seleção pelo usuário. Neste caso, foi modelado o *PresentationFormat* com apenas um *ConceptDisplayUnit* para os possíveis valores que fazem parte do conceito (Modo de ventilação e tipo de secreção respiratória apresentada pelo paciente). Na segunda linha, cada *PresentationFormat* do *PresentationSet* foi formatado do mesmo modo, já que se tratam do mesmo assunto (configuração de parâmetros ventilatórios para ventilação mecânica do paciente). A modelagem dos indivíduos foi semelhante aos *PresentationFormat* da Figura 13 (página 53), exceto pelas seguintes alterações:

- Omissão dos ícones, usando apenas dois *ConceptDisplayUnit*: um para o valor do conceito, que não segue o padrão *composite* neste caso, e outro para exibir a unidade.
- Uso da restrição do *layout* do conteúdo na vertical (campo de entrada acima da unidade), através do relacionamento *hasSpatialConstraint* com o indivíduo _contentLayoutVertical, conforme Figura 42, para o parâmetro ventilatório frequência respiratória.

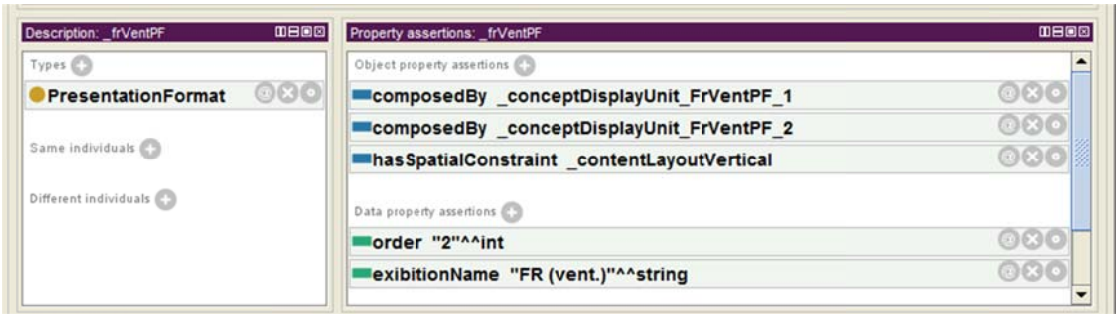

Figura 42 - Definição de PresentationFormat para parâmetro ventilatório

Outros resultados estão disponíveis na seção 5.3, Comparação com trabalhos relacionados (página 92).

5.2 Aspectos relacionados à modelagem ontológica

Para o trabalho desenvolvido, ressaltamos que, mesmo para conceitos biomédicos simples utilizados como exemplos, existem dezenas de configurações de *layout* possíveis que devem ser identificadas, modeladas e implementadas, o que torna o sistema complexo e de difícil modelagem, especialmente com a identificação e inclusão de conceitos mais elaborados, fluxo de trabalho, uso de terminologias médicas etc. E quanto mais complexa a modelagem, mais genérico deve ser o algoritmo para construção da interface, o que é necessário para evitar alterações corriqueiras de código-fonte em ambiente de produção, o que está fora dos objetivos deste trabalho, já que a ideia é permitir inserção de novos documentos com *layouts* sem necessidade de recompilação e reimplantação da aplicação no servidor.

Até o estágio atual do presente estudo, optamos pela modelagem relacionada ao padrão de projeto *decorator* para o *layout* dos formatos de apresentação, pois cada *ConceptDisplayUnit* funciona como um *wrapper*, seja para o valor do conceito biomédico que exibe, seja para símbolos, rótulos, unidades de medida, incluindo metadados para *layout* e estilo, conferindo, deste modo, não novas funcionalidades como na definição original do padrão, mas novas características visuais. O *PresentationFormat* funciona, deste modo, como uma espécie de *wrapper*-raiz. A modelagem inclui também abordagem semelhante ao padrão *composite* no caso de *ConceptDisplayUnit*, pois agrega itens correlacionados semântica e visualmente.

Os conceitos biomédicos, entretanto, são mantidos, puros sem informações de representação gráfica, o que é crucial para o intercâmbio deste tipo de informação. Foi realmente necessário o uso de orientações espaciais para ordenação dos *ConceptDisplayUnit*, de modo que cada orientação dessas possa ser representada por um componente genérico de UI, no caso, uma tabela ou um sistema de *grid*.

A modelagem da classe *PresentationFormat* refere-se, portanto, ao último estágio da apresentação da informação ao usuário e tal apresentação, além da própria informação apresentada, requer toda a sua estruturação espacial e de estilo. Adotamos uma abordagem *bottom-up*, partindo da menor unidade de exibição de dados até a montagem completa do formato.

Diante do exposto, ela representa uma espécie de sub-ontologia ou um modelo de apresentação de conteúdo com abordagem relacionada à descrita como *Display Content Unit* (VAN DER LINDEN et al., 2009) para representação visual dos arquétipos.

A classe *PresentationFormat* pode ainda ser especializada em outras com itens semânticos biomédicos em comum e, portanto, dimensionamento e posição de componentes idênticos.

As seções a seguir tratam de realizar um breve comparativo entre o presente estudo e os diversos trabalhos relacionados encontrados na literatura sobre geração automática e a modelagens de UI/*layouts* em geral, não específicas a nenhum domínio de aplicação. Notou-se que alguns identificaram ou implementaram características de um gerador de UI/ *layout* que outros não o fizeram. Foram encontradas, portanto, diversas abordagens e arquiteturas adotadas para uma mesma finalidade. Observou-se também que a geração de *layouts* de UI para documentos fazem parte da geração de interfaces gráficas, não sendo comum encontrar trabalhos isolados para geração de *layout* de documentos.

5.3 Comparação com trabalhos relacionados

Para o presente trabalho, foram realizadas algumas comparações, tanto relacionadas à abordagem adotada quanto aos resultados, com trabalhos encontrados na literatura e em especial com o trabalho original de (DUARTE, 2011).

5.3.1 OpenCTI/MedViewGen

Tomamos um exemplo do trabalho original de (DUARTE, 2011), onde é exibido o conteúdo do formulário de Certificado de Óbito, Aba *Ocorrência*. Nesta figura, observam-se muitos espaços vazios e campos do desordenados. Este formulário foi remodelado e gerado utilizando-se da abordagem do presente estudo e na Figura 43 exibimos um comparativo.

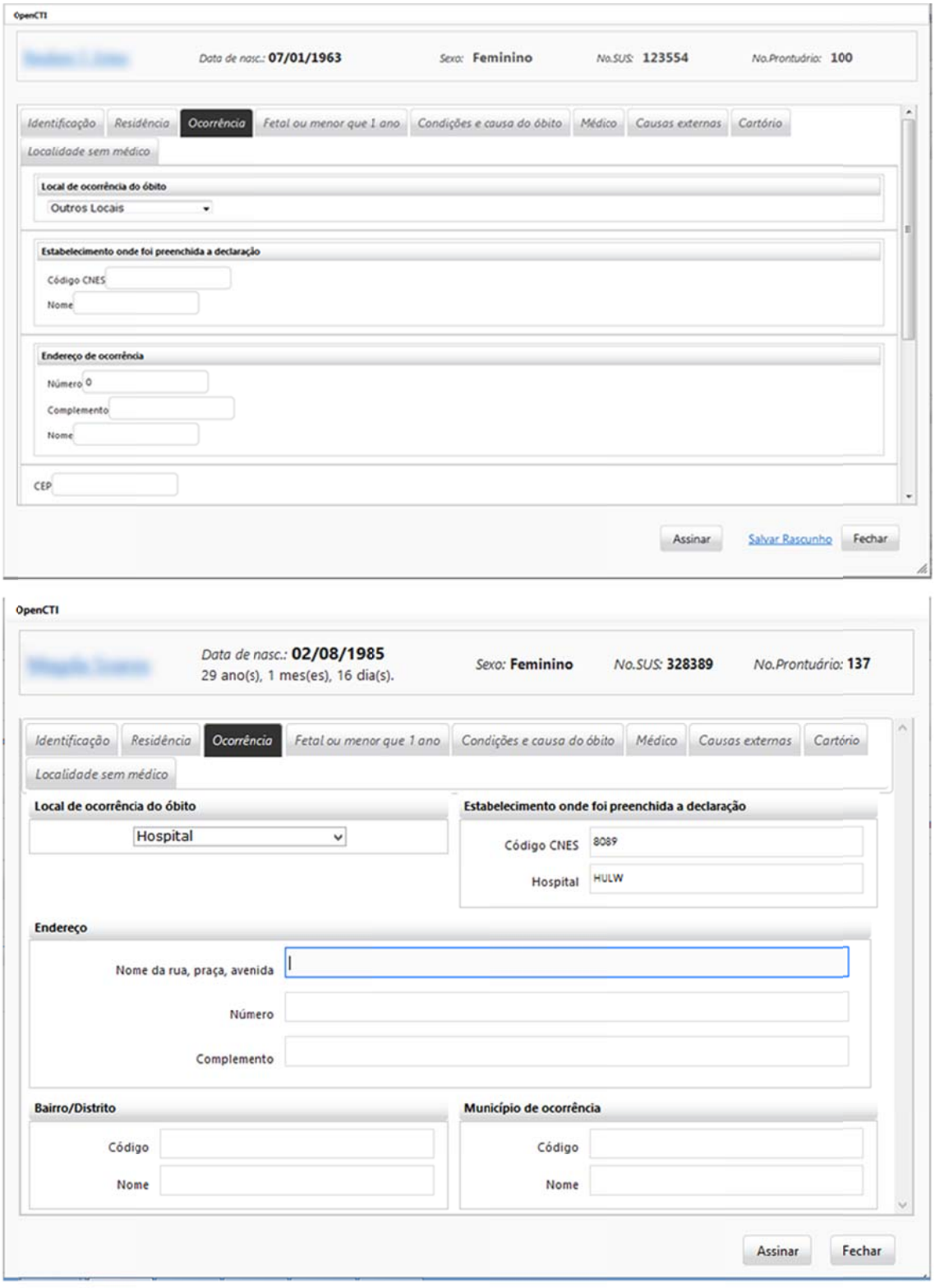

Figura 43 - Comparação do *layout* do formulário Certificado de óbito (aba Ocorrência) do trabalho original de (DUARTE, 2011) com o presente estudo

Para o formulário da primeira tela da Figura 43*,* correspondente à UI gerada no trabalho *MedViewGen* (DUARTE, 2011), na aba *Ocorrência*, o endereço de ocorrência (do óbito) havia sido modelado como um arquétipo (classe *Archetype*) contendo um *AbstractBiomedicalConcept,* conceito biomédico abstrato (endereço de ocorrência) com três relacionamentos do tipo *hasPart* com outros conceitos mais simples (nome da rua, número e complemento), cujos campos são exibidos desordenados. Para o formulário da segunda tela da mesma figura, correspondente ao *layout* gerado para a UI no presente trabalho, foi necessária uma remodelagem para cada conceito biomédico com uso de *PresentationFormat* e *ConceptDisplayUnit*, em lugar da classes legadas *Archetype* (DUARTE, 2011) *e AbstractBiomedicalConcept* (NÓBREGA, 2010), especialmente para atribuir a ordem de cada campo do conceito no formulário.

Nos componentes do tipo *panel* (caixas com barra de título cinza) foi inserido pelo algoritmo desenvolvido neste estudo um pequeno *grid*, configurado na ontologia de documentos como restrição espacial através do relacionamento *hasSpatialConstraint* para cada *PresentationFormat* desta seção do documento, de modo a organizar o seu conteúdo como *layout* de formulário (os rótulos alinhados à direita e campos de entrada de texto alinhados à esquerda e com mesma largura). O endereço de ocorrência foi modelado como *PresentationSet* composto apenas de um *PresentationFormat*, para que ocupasse a largura da seção do documento, já que é comum que nomes de ruas sejam longos. Por razões estéticas, foi permitido que os demais campos *número* e *complemento* tivessem a mesma largura. Nitidamente, houve melhora da aparência do documento atual em relação ao *layout* do documento do trabalho original (DUARTE, 2011).

Pode-se observar, no entanto, que se na linguagem OWL houvesse a definição de propriedades (*data properties*) para relacionamentos, seria possível atribuir um *dataProperty order* do tipo inteiro no próprio *objectProperty hasPart* do conceito composto *AbstractBiomedicalConcept*, simplificando a modelagem sem necessidade de classes adicionais para este fim.

A Figura 44 exibe outra comparação. A primeira tela refere-se à aba Identificação, do Certificado de Óbito, presente no trabalho original (DUARTE, 2011) e a segunda, ao *layout* da UI gerado no presente trabalho.

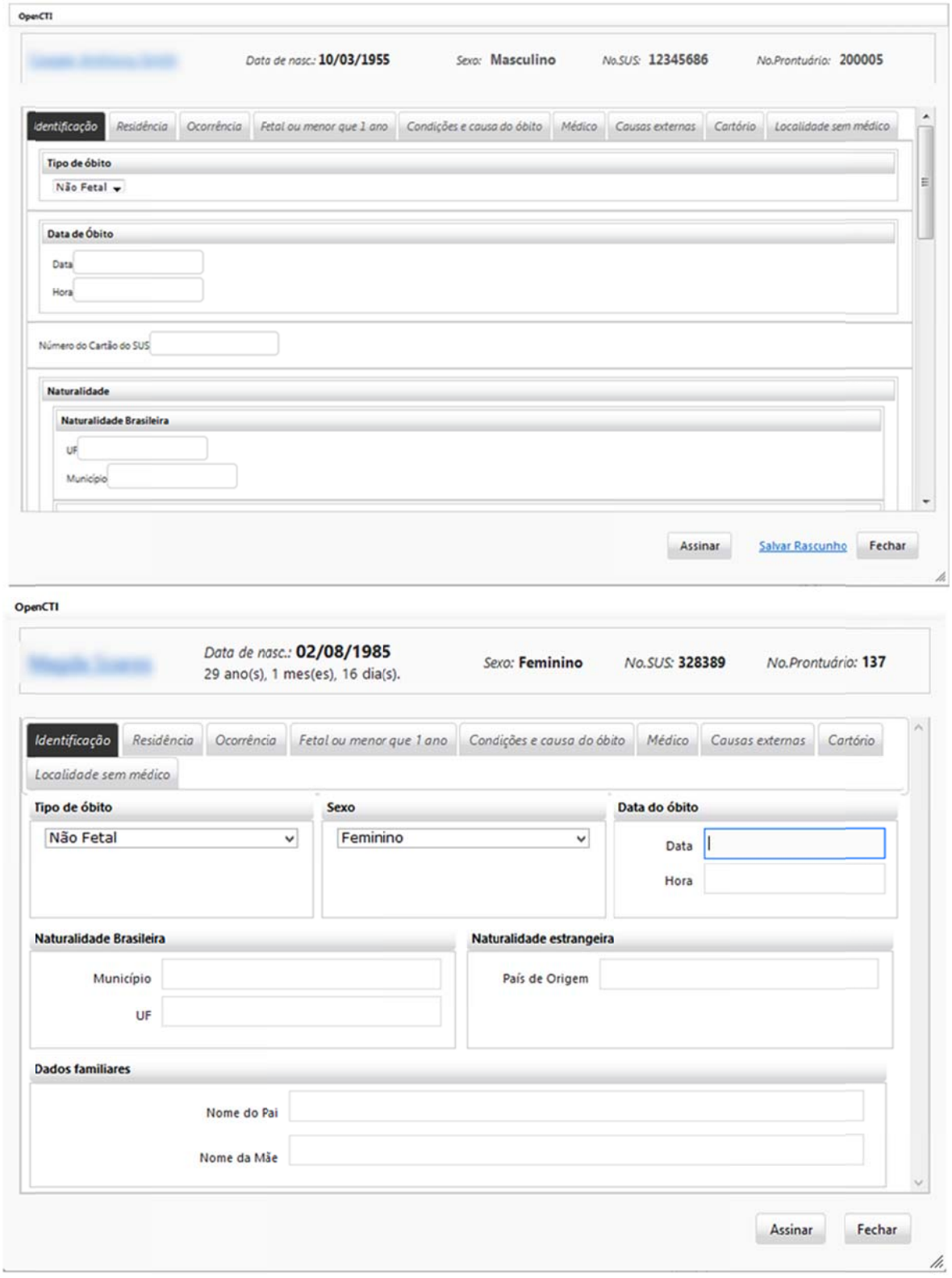

Figura 44 - Comparação do *layout* do formulário Certificado de óbito (aba Identificação) do trabalho original de (DUARTE, 2011) com o presente estudo

Outro exemplo de comparação pode ser verificado a seguir. Conforme visto nos exemplos anteriores, a definição de arquétipo do projeto original, o MedViewGen (DUARTE, 2011), não impõe uma restrição sequencial lógica dos conceitos menores que o compõem, neste caso, o arquétipo “Pressão arterial”, conforme Figura 45 (à esquerda, exibição de captura de tela parcial), que apresenta campos desordenados. A ordem correta ou usual, de acordo com as convenções médicas, seria a exibição dos valores: pressão sistólica, separador, pressão diastólica.

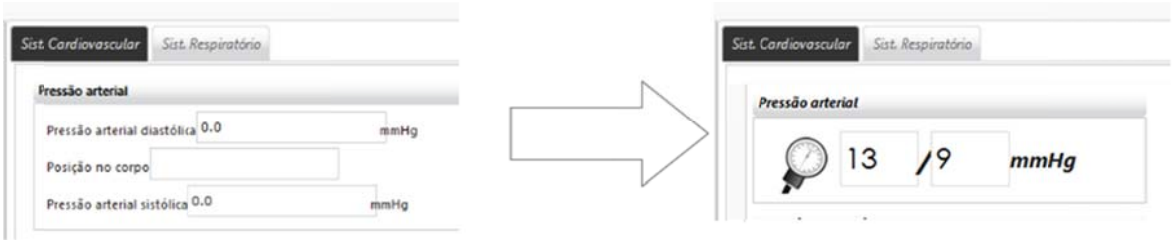

Figura 45 - Comparação entre *layouts* do conceito de pressão arterial

No *layout* obtido para o formato de apresentação (Figura 45, à direita), o atributo *Posição do corpo* foi omitido para o conceito formatado, por simplicidade, já que é pouco utilizado na prática. Além da ordem trocada dos campos na Figura 45 (à esquerda), outra observação quanto a este exemplo se refere ao tamanho dos componentes de entrada de texto, que estão demasiadamente largos para o valor a ser inserido considerando o formato de apresentação compacto proposto para o conceito de pressão arterial. As dimensões dos campos foram diminuídas com atribuição de valores máximos, ainda que arbitrários, no modelo ontológico de conceitos biomédicos, conforme mencionado na seção 3 (página 47). Para o exemplo em questão, considerou-se o valor máximo de 300 mmHg (arbitrário), o que implica, na prática, em um tamanho máximo do componente igual a três caracteres, mais um adicional para a casa decimal. Contudo, o *ConceptDisplayUnit* identificado e modelado requer uma transformação para exibição desse dado, que é a divisão por dez do valor do conceito, ou seja, se a pressão sistólica obtida foi 130 mmHg, será representado na tela pelo valor 13, comumente usado sem casas decimais, contribuindo, deste modo, para a diminuição do tamanho do componente. Com essa restrição, economiza-se espaço em tela, o que é útil para exibição em dispositivos pequenos e ainda melhora a estética do formulário neste caso.

Portanto, observa-se que os formulários gerados no trabalho original (DUARTE, 2011) não possuem organização formal em *grid*, de modo que os contêineres de cada arquétipo ocupam todo o espaço livre na horizontal, prejudicando a estética, deixando áreas vazias em excesso, além da falta de ordenação dos campos e desalinhamento dos rótulos. No presente estudo, foi alterado o modelo de documentos e definida uma formatação em *grid* que melhorou a aparência dos documentos como um todo.

5.3.2 Metawidget

O Metawidget (KENNARD; LEANEY, 2010), descrito na seção 2.3.5.3, é um gerador de automático de UI com *layout* e estilo e está disponível para *download* sob licença gratuita ou comercial. O Quadro 5 enumera algumas características importantes para comparação do Metawidget com o presente trabalho. Optamos inserir o *MedViewGen* (DUARTE, 2011) neste quadro já que o mesmo corresponde ao gerador automático de UI e o presente estudo ao seu complemento, ou seja, a geração de *layout*/estilo.

	Metawidget (KENNARD; LEANEY, 2010) (KENNARD, 2014)	**MedViewGen** (DUARTE, 2011)	**Trabalho atual**
Escopo	Geração de UI/*layout* e estilo de formulários	Geração de UI de formulários	Geração de *layout*/estilo de formulários
Modelagem semântica (em ontologia) de componentes de UI e/ou *layout*/estilo	Não	Sim	Sim
Busca de metadados para geração de UI e/ou *layout*/estilo	Sim, vem com múltiplos inspetores específicos para cada tipo de fonte a ser inspecionada à procura de metadados para geração de UI (POJO, XML etc).	Sim, em ontologias OWL, mas sem informações específicas de *layout* e estilo.	Sim, em ontologias OWL é realizada a busca de informações sobre *layout* e estilo.

Restrições abstratas	Ordenação de campos de formulários com anotação Java (@UiComesAfter) para atributos dos objetos ou configuração externa em XML.	Ordenação de arquétipos dentro da seção do documento através de atributo na ontologia.	Ordenação de formatos de apresentação dentro da seção do documento através de atributo na ontologia e ordenação dentro de cada formato de apresentação.
Divisão do formulário em seções	Configurada com anotação Java (@UiSection) no objeto de domínio.	Determinada na ontologia OWL com indivíduos associados ao corpo do documento.	Não.
Definição de grid	Sim, para cada seção do formulário, em código-fonte ou em XML.	Não usa grid nas seções dos formulários.	Sim, para cada seção do formulário, na ontologia OWL.
Distribuição do conteúdo no grid	Segue a sequência da esquerda para a direita até o limite do número de colunas.	Não usa grid.	Segue a sequência da esquerda para a direita até o limite do número de colunas. Uso de *PresentaionSet* para *grids* esparsos..
Escolha de componentes com mesma função na UI	Sim, em código-fonte ou em XML.	Não implementado.	Não.
Processamento de estilo CSS	Sim	Não	Sim
Configuração de componente de UI para texto longo	Configurado com anotação Java (@UiLarge) no objeto de domínio ou através de XML .	Não.	Inserido atributo para tamanho máximo de *string* na ontologia do conceito biomédico respectivo e, a partir disto, atribuição do componente de entrada de texto longo.
Objetos imutáveis (após instanciação)	Sim	Sim	Sim, no caso, indivíduos, que podem ser alterados somente pelo desenvolvedor para reconfiguração.

Suporte a diversas tecnologias	Sim, com implementação para várias tecnologias.	Sim, porém com implementação apenas para JSF até o momento.	Depende do trabalho base elaborado por (DUARTE, 2011). O *layout* e estilo do presente estudo se aplicam a interfaces *web*, mas pode ser realizado mapeamento para outras tecnologias.

Quadro 5- Comparação entre trabalhos relacionados à geração de UI e/ou *layout*/estilo

Alguns itens merecem destaque em relação ao Metawidget. O desenvolvedor que desejar instalá-lo em uma aplicação existente deve criar um arquivo de configuração em XML com esquema definido pelo próprio Metawidget para incluir todas as restrições necessárias para a geração da UI/*layout*. Opcionalmente é possível inserir anotações Java em código-fonte. Os inspetores embutidos do Metawidget podem, portanto, extrair metadados para geração da UI/*layout* de arquivos com estrutura previamente conhecida, como classes Java ou o arquivo XML definido pelo próprio gerador de UI.

Com relação à modelagem, não há definição explícita ou formalmente descrita de modelos de conceitos de componentes de UI/*layout* ou *templates* de documentos para os formulários, que são definidos e construídos através da inspeção de objetos de domínio previamente existentes e configurações. Em termos de consolidação do conhecimento, acreditamos ser mais adequada a modelagem ontológica dos domínios necessários para a geração de interfaces/ *layout*.

A Figura 46 exibe um exemplo de formulário gerado pelo Metawidget para um objeto Java da classe Person, que representa uma pessoa com seus dados pessoais.

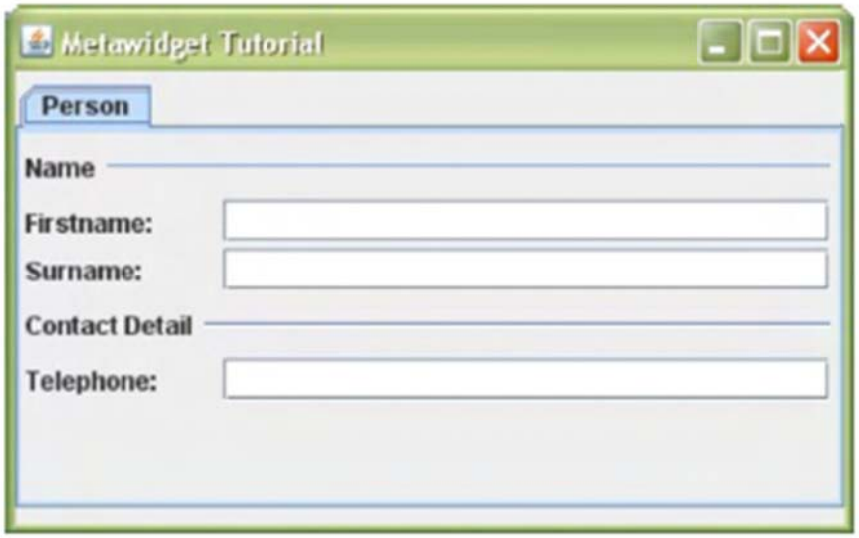

Figura 46 - Exemplo de formulário gerado pelo Metawidget (KENNARD; LEANEY, 2010)

Portanto, a abordagem entre os trabalhos é diferente no que diz respeito à modelagem, mas possui muitos pontos em comum para soluções específicas de *layouts*. O *Metawidget*, no entanto, é uma aplicação já consolidada com suporte nativo a diversas tecnologias de UI e vem sendo utilizado por muitos desenvolvedores.

5.3.3 Abordagem de dois modelos

Um breve resumo deste trabalho se encontra na seção 2.3.5.2 e o seu objetivo geral é a criação de um *framework* para alcançar a interoperabilidade no nível de apresentação para sistemas de RES.

Considerando especificamente o *layout* de UI desenvolvido no trabalho, os autores afirmam que no modelo não foi possível definir uma exibição genérica sem escolher um elemento de tecnologia específico com HTML ou <table>, sendo, entretanto, parte do projeto em geral e de restrições de exibição, não do conteúdo.

No presente trabalho optamos por usar uma abstração quanto ao *layout* na classe *PresentationFormat* e itens relacionados, definindo apenas a orientação espacial dos itens filhos do tipo *ConceptDisplayUnit.* Esta orientação foi incorporada

no próprio *ConceptDisplayUnit* e *PresentationFormat*, incluída como restrição, fornecendo informação para orientação espacial do *grid* interno da estrutura, atribuído automaticamente e modelado na ontologia respectiva como *conceptGroup* (agrupador de elementos de conceito). Isto permitiu isolar definições de componentes de UI do *ConceptDisplayUnit*, deixando a atribuição de componentes a cargo do algoritmo AAC.

Neste caso, o processamento insere automaticamente, para um *conceptGroup*, um componente genérico na árvore de UI do tipo *panelGrid*, para organizar os elementos na direção horizontal (uma linha), na vertical (uma coluna) ou em formulário (duas colunas com alinhamento dos rótulos). Em termos de tecnologia JSF, é um objeto do tipo *javax.faces.component.html.HtmlPanelGrid*, que é renderizado como HTML <table>. Optamos pelo uso de tabela por simplicidade, para evitar mais aninhamentos de elementos .

Em nossa modelagem ontológica do *layout*, se tivéssemos definido diretamente uma estrutura de componente (ex.: *table*), estaríamos alterando a filosofia original de todo o projeto OpenCTI com relação à atribuição automática de componentes, fazendo em parte o serviço do algoritmo antes que o mesmo fosse executado.

Observamos, entretanto, que é realmente muito difícil, se não impossível, uma definição de formatos de apresentação para *layout* sem que haja menção a algum item estrutural ou restrição para orientar a disposição espacial dos dados ou elementos na UI.

Ressaltamos uma semelhança entre as modelagens, pois constatamos a necessidade da classe *ConceptDisplayUnit* na ontologia de documentos, descrita em OWL, assim como no trabalho de (VAN DER LINDEN et al., 2009) foi desenvolvido conceito de DisplayContentUnit (DCU), exemplificado na Figura 5, página 39, em XML. Observa-se também que neste exemplo de DCU não há especificação de *restrições* espaciais para o *layout* ou mesmo estilo (tipo de fonte, cores etc.). No trabalho (artigo) não foram encontrados capturas de tela das interfaces gráficas geradas.

6

Considerações Finais

Os documentos do prontuário do paciente são desenvolvidos, muitas vezes, com base em sua versão em papel, o que não permite completa utilização do potencial dos meios eletrônicos.

Os formatos de apresentação de *layout* para o RES, desenvolvidos no presente estudo, se padronizados, uniformizam o aspecto visual para o(s) conceito(s) biomédico(s) que formatam, tornando mais consistente a interface gráfica, mantendo a mesma aparência entre os documentos. Convenções visuais bem planejadas e estabelecidas para documentos do RES facilitam, inclusive, a localização da informação na tela (GALVÃO; RICARTE, 2012), o que pode melhorar a usabilidade do sistema. Este trabalho contribuiu neste sentido, alcançando seu Objetivo 1 (página 19) ao propor uma modelagem ontológica apropriada para o desenvolvimento destes formatos específicos, que podem ser reusáveis, não somente ao *layout* genérico do documento como um todo.

Em relação ao trabalho original (DUARTE, 2011), conseguimos melhorar a estética da UI, alcançando o Objetivo 2, ordenando os itens dentro dos arquétipos através das restrições espaciais impostas nos formatos de apresentação e organizando os elementos da interface em um *grid*, conferindo-lhes uma distribuição espacial mais uniforme. Constatamos, portanto, que há viabilidade no uso da técnica empregada para geração do *layout*, levando em consideração as limitações encontradas (seção 1.3, página 19).

Os objetivos 3 e 4 foram alcançados com a extensão da API original com inserção e modificação de algoritmos existentes no trabalho original (DUARTE, 2011) e integração as solução ao sistema OpenCTI.

Ressaltamos que o uso de ontologias OWL permitiu que certos aspectos das tecnologias e especificações envolvidas fossem encapsulados (CSS, JSF), tornando possível atribuir configurações de *layout* e estilo aos documentos do RES apenas instanciando ou associando indivíduos já existentes na ontologia respectiva. Com auxílio do *software* Protégé é possível a edição de arquivos OWL, entretanto a sua interface gráfica requer melhorias de usabilidade.

O gerador de interface gráfica do trabalho original (DUARTE, 2011) e sua versão com *layout* e estilo desenvolvida neste trabalho, a princípio, se aplicam a documentos de qualquer domínio de negócios, desde que estejam modelados nas respectivas ontologias os conceitos inerentes à área e os modelos dos documentos.

A abordagem do presente trabalho juntamente com o trabalho original, portanto, teriam maior aplicação como estudo avançado para o desenvolvimento de novas abordagens e arquiteturas de sistemas para a *web* semântica.

6.1 Trabalhos futuros

Inúmeras possibilidades podem surgir durante o desenvolvimento e a conclusão de um trabalho. A área de informática em saúde está em franca expansão, mas depende inicialmente, do desenvolvimento da informática e da computação em si. A seguir, comentamos sobre os possíveis trabalhos futuros a partir do conhecimento e dos resultados obtidos no presente estudo.

6.1.1 Modelagem de novos formatos de apresentação

Os modelos de formatos de apresentação propostos até o estágio atual do presente estudo se limitam a conceitos biomédicos com dados comuns (texto, listas para seleção). Outros tipos de dados podem ser explorados, como gráficos, imagens etc, que requerem componentes de interface gráfica e conceitos biomédicos mais complexos e mesmo definição ontológica de comportamento de UI.

Ainda é possível implementar várias especializações da classe *PresentationFormat*, de modo que, de acordo com a subclasse, o *layout* seja automaticamente aplicado. Como exemplo, podemos ter uma subclasse *VitalSignPresentationFormat*, que seria o formato para os conceitos envolvendo exibição de sinais vitais. Esta classe pode ter uma formatação padrão definida na ontologia, tornando mais fácil a instanciação de novos indivíduos.

6.1.2 Usar outras especificações técnicas para *layout*

A maneira de construir *layouts* de páginas *web* é atualmente baseada no Box Model do CSS 2 (W3C WORLD WIDE WEB CONSORTIUM, 2011) e, ainda, no uso de tabelas HTML. A metodologia conhecida como *Tableless*, entretanto, se utiliza apenas do Box Model – excluindo-se as tabelas – para definição de *layouts*. É uma técnica que requer alteração em várias propriedades CSS dos elementos na página para configuração de um *layout* simples.

O W3C está redigindo uma especificação, ainda em andamento, chamada *CSS Grid Layout Module Level 1* (W3C WORLD WIDE WEB CONSORTIUM, 2014), que altera radicalmente o modo de construção de *layouts* web, deixando de lado o Box Model, definindo um sistema de *layout* baseado em *grids* de duas dimensões. Na proposta, os elementos de um *grid* podem ser posicionados arbitrariamente em *slots* com uso da linguagem CSS, sem a necessidade de declarar as propriedades de posicionamento para cada elemento DIV da página com vários níveis de aninhamento (*float, position, top, bottom, clear* etc).

A especificação parece promissora e, apesar de não ter sido lançada, é interessante observar a técnica e estudar maneiras de incorporá-la ao gerador de *layout* no futuro.

6.1.3 Refatoração

Inicialmente, a remodelagem foi pensada de modo que levasse a um número mínimo de alterações no algoritmo AAC já implementado, o que não foi possível. O assunto abordado é extremamente diversificado, envolvendo vários modelos

ontológicos de áreas complexas, várias tecnologias e, portanto, várias configurações condicionais dos componentes de UI que devem ser aplicadas através de linguagem de programação. E quanto mais se atualizam os modelos ontológicos, mais metadados para *layout* são incorporados, e, por isso, mais processamento deve ser realizado. Portanto, para evitar um encadeamento de inúmeras estruturas de decisão *if ... then ... else,* o ideal seria uma refatoração de todas as API envolvidas usando alguma técnica de programação declarativa ou ainda tabelas de decisão semânticas para facilitar a codificação. Com isso, podem ser evitadas consequências futuras de impacto negativo na legibilidade e manutenção de código-fonte.

REFERÊNCIAS

AGUIAR, A.; SOUSA, A.; PINTO, A. **User-Case Controller**. 2001. Disponível em: <http://www.cs.sjsu.edu/~pearce/modules/lectures/ooa/references/usecases/Aguiar_al_2001_ART_Use_Case_Controller.pdf>Acesso em: jan. 2013.

BACELAR, S. **O que é Prontuário Médico ?.** 2012. Disponível em: <http://www.hub.unb.br/noticias/artigos/prontuariomedico.html> Acesso em: 11 nov. 2012.

BRASIL. **Constituição (1988). Constituição da República Federativa do Brasil**. Brasília, DF: Senado Federal. 1988. Disponível em <http://www.planalto.gov.br/ccivil_03/constituicao/constituicaocompilado.htm>. Acesso em: 10 mai. 2014.

CERNY, T.; CHALUPA, V.; DONAHOO, M. J. Impact of User Interface Generation on Maintenance. *IEEE*, 2012. pp. 621-625.

CONSELHO FEDERAL DE MEDICINA. **Resolução 1.821/2007.** 2007. Disponível em: <http://www.sbis.org.br/> Acesso em: 10 nov. 2012.

_____. **Resolução 1.931/2009.** 2009. Disponível em: <http://portal.cfm.org.br/index.php?option=com_content&view=category&id=9&Itemid=122> Acesso em: 10 mai. 2014

_____. **Cartilha sobre Prontuário Eletrônico:** A Certificação dos Sistemas de Registro Eletrônico em Saúde*.* 2012. Disponível em: <http://www.sbis.org.br/certificacao/Cartilha_SBIS_CFM_Prontuario_Eletronico_fev_2012.pdf> Acesso em: 10 nov. 2012.

D'AQUIN, M.; NOY, N. F. Where to publish and find ontologies? A survey of ontology libraries. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2012. pp. 96-111.

DUARTE, R. C. M. **MedViewGen**: Uma Ferramenta Baseada em Ontologias para Geração Automática de Interface com o Usuário para Documentos em Registros Eletrônicos em Saúde. 2010. 171f.. Dissertação (Mestrado em Informática). Universidade Federal da Paraíba, João Pessoa, 2010.

FALB, J. et al. Fully-automatic generation of user interfaces for multiple devices from a high-level model based on communicative acts. *Proceedings of the 40th Hawaii International Conference on System Sciences*, 2007. pp. 1530-1605.

GALVÃO, M. C. B.; RICARTE, I. L. M. **Prontuário do Paciente**. Rio de Janeiro: Guanabara Koogan, 2012.

GREENES, R. **Clinical Decision Support:** The Road Ahead. San Diego: Elsevier, 2006.

HAYWOOD, D. **Domain-Driven Design Using Naked Objects**. s.l.:Pragmatic Bookshelf, 2009.

HERVÁS, R.; BRAVO, J. Towards the ubiquitous visualization: Adaptive user-interfaces based on the Semantic Web. *Interacting with Computers*, 2011. pp. 40-56.

INSTITUTE OF MEDICINE. **Key Capabilities of an Electronic Health Record System**. 2003. Washington D.C.: The National Academies Press. 2003. Disponível em <http://books.nap.edu/openbook.php?record_id=10781> Acesso em 20 mai. 2014.

JONES, C. et al. A metadata-driven framework for generating field data entry interfaces in ecology. *Ecological Informatics*, 2007. pp. 270-278.

KENNARD, R. **Metawidget User Guide and Reference Documentation.** 2014. Disponível em: <:http://metawidget.sourceforge.net/doc/reference/en/html/index.html> Acesso em: 13 mai. 2014.

KENNARD, R.; LEANEY, J. Towards a general purpose architecture for UI generation. *The Journal of Systems and Software*, 2010. pp. 1896-1906.

KENNARD, R.; LEANEY, J. Is there convergence in the field of UI generation?. *The Journal of Systems and Software*, 2011. pp. 2079-2087.

LO, H. et al. Electronic health records in specialty care: a time-motion study. *Journal of the American Medical Informatics Association*, 2007. pp. 609-615.

LOK, S.; FEINER, S. **A Survey for Automated Layout Techniques for Information Presentations.** 2001. Disponível em: <http://www1.cs.columbia.edu/~lok/papers/layoutsurvey.pdf> Acesso em: 10 nov. 2012.

MASSAD, E.; MARIN, H. D. F.; AZEVEDO NETO, R. S. D. **O prontuário eletrônico do paciente na assistência, informação e conhecimento médico**. São Paulo: H. de F. Marin. 2003.

MOZILLA DEVELOPER NETWORK. *XUL (XML User Interface Language).* 2005. Disponível em: <https://developer.mozilla.org/en-US/docs/XUL> Acesso em: 10 jan. 2013.

NÓBREGA, H. I. **Framework para modelagem e utilização de formulários médicos e termos clínicos utilizando OWL.** 2010. 51f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação). Universidade Federal da Paraíba, João Pessoa, 2010.

OASIS. **User Interface Markup Language (UIML) 4.0.** 2009. Disponível em: <:docs.oasis-open.org/uiml/v4.0/uiml-4.0.pdf> Acesso em: 13 dez. 2012.

OBJECT MANAGEMENT GROUP. **Unified Modeing Language (UML) 2.4.1.** 2011. Disponível em: <:http://www.omg.org/spec/UML/2.4.1/> Acesso em: 29 dez. 2012.

ORACLE. **Java Server Faces Technology**. 2012. Disponível em: <:http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html> Acesso em: 22 dez. 2012.

PAWSON, R. **Naked Objects**. 2004. 223f. Tese (Doutorado em Filosofia). Universidade de Dublin, Irlanda, 2004. Disponível em:

<http://downloads.nakedobjects.net/resources/Pawson%20thesis.pdf>
Acesso em: 06 jan. 2013.

PREECE, J.; ROGERS, Y.; SHARP, H. **Interaction Design**: beyond human-computer interaction. John Wiley & Sons, 2002.

SONNINO, R.; SONNINO, B. **Introdução ao WPF**. 2012. Disponível em: <http://msdn.microsoft.com/pt-br/library/cc564903.aspx> Acesso em: 10 jan. 2013.

STANFORD UNIVERSITY. **Protégé**. *2014.* Disponível em: <http://protege.stanford.edu/about.php> Acesso em: 20 mai. 2014.

TORTOSA, M.; MARTÍNEZ-COSTA, C.; FERNÁNDEZ-BREIS, J. A Generative Tool for Building HealthApplications Driven by ISO 13606 Archetypes. *J Med Systems*, 2012. pp. 3063-3065.

VAN DER LINDEN, H.; AUSTIN, T.; TALMON, J. Generic screen representations for future-proof systems, is it possible? There is more to a GUI than meets the eye. *Computer Methods and Programs in Biomedicine*, 2009. pp. 213-226.

W3C WORLD WIDE WEB CONSORTIUM. **Cascading Style Sheets, level 1**. 1996. Disponível em: <http://www.w3.org/TR/REC-CSS1-961217> Acesso em: 20 mai. 2014.

_____. **HTML 4.01 Specification.** 1999. Disponível em: <http://www.w3.org/TR/html401/> Acesso em: 20 mai. 2014.

_____. **Document Object Model (DOM) Level 2**: HTML Specification. 2003. Disponível em: <http://www.w3.org/TR/DOM-Level-2-HTML/> Acesso em: 20 jul. 2014.

_____. **The Forms Working Group.** 2011. Disponível em: <http://www.w3.org/MarkUp/Forms/#waXForms> Acesso em: 06 jan. 2013.

_____. **OWL 2 Web Ontology Language**. 2012. Disponível em: <http://www.w3.org/TR/owl2-overview/> Acesso em: 29 dez. 2012.

_____. **CSS Grid Layout Module Level 1.** 2014. Disponível em: <http://www.w3.org/TR/2014/WD-css-grid-1-20140513/> Acesso em: 20 mai. 2014.

APÊNDICE I

Este mapeamento foi criado para o presente trabalho para uso nas ontologias de UI e mapeamento para tecnologia concreta de UI. Os nomes genéricos das propriedades foram escolhidos por semelhança com especificação de *layout* e estilo (CSS) para facilitar memorização, Contudo, para diferenciar quando trabalhamos com interface genérica ou concreta, usamos atributos genéricos com nomes discretamente diferentes com grafia em *camelCase* em vez do hífen, até mesmo para visualizar melhor o mapeamento. O dicionário não abrange todas as possíveis propriedades mapeáveis para tecnologia concreta e, ainda, as propriedades genéricas criadas serviram apenas como ponto de partida para atribuir *layout* e estilo aos exemplos de interfaces gráficas usados neste trabalho.

Nome da propriedade genérica	**Categoria**	**Equivalente em CSS 2**
bgColor	Estilo	background-color
borderColor	Estilo	border-color
fontTypo	Estilo	font-family
fontSize	*Layout*	font-size
fontWeight	Estilo	font-weight
fontColor	Estilo	color
Margin	*Layout*	margin
Padding	*Layout*	padding
textAlignment	*Layout*	text-align
Width	*Layout*	Width

ANEXO I

Neste anexo foram incluídas duas figuras adicionais, Figura 47 e Figura 48, originais do trabalho MedViewGen (DUARTE, 2011).

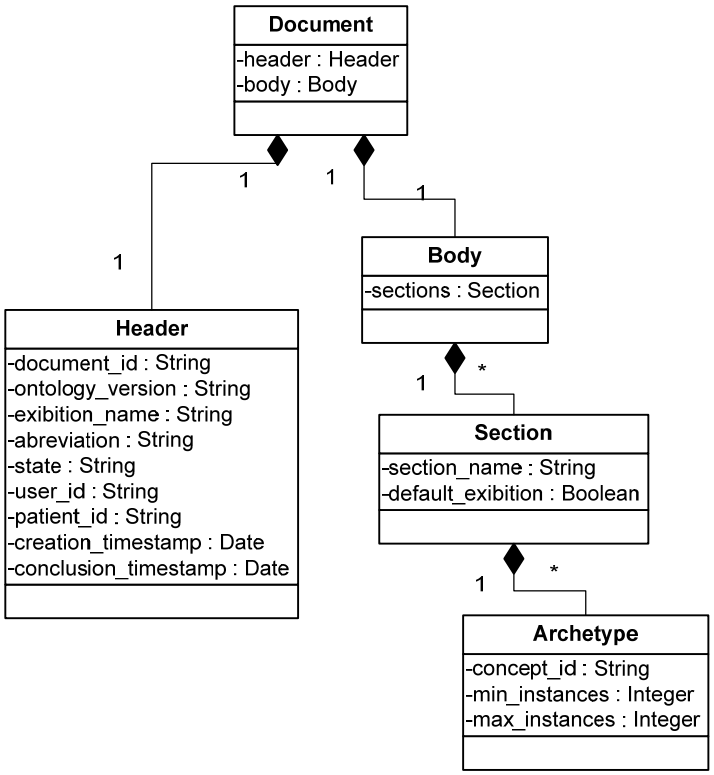

Figura 47 - Modelo ontológico de documentos clínicos do MedViewGen (DUARTE, 2011).

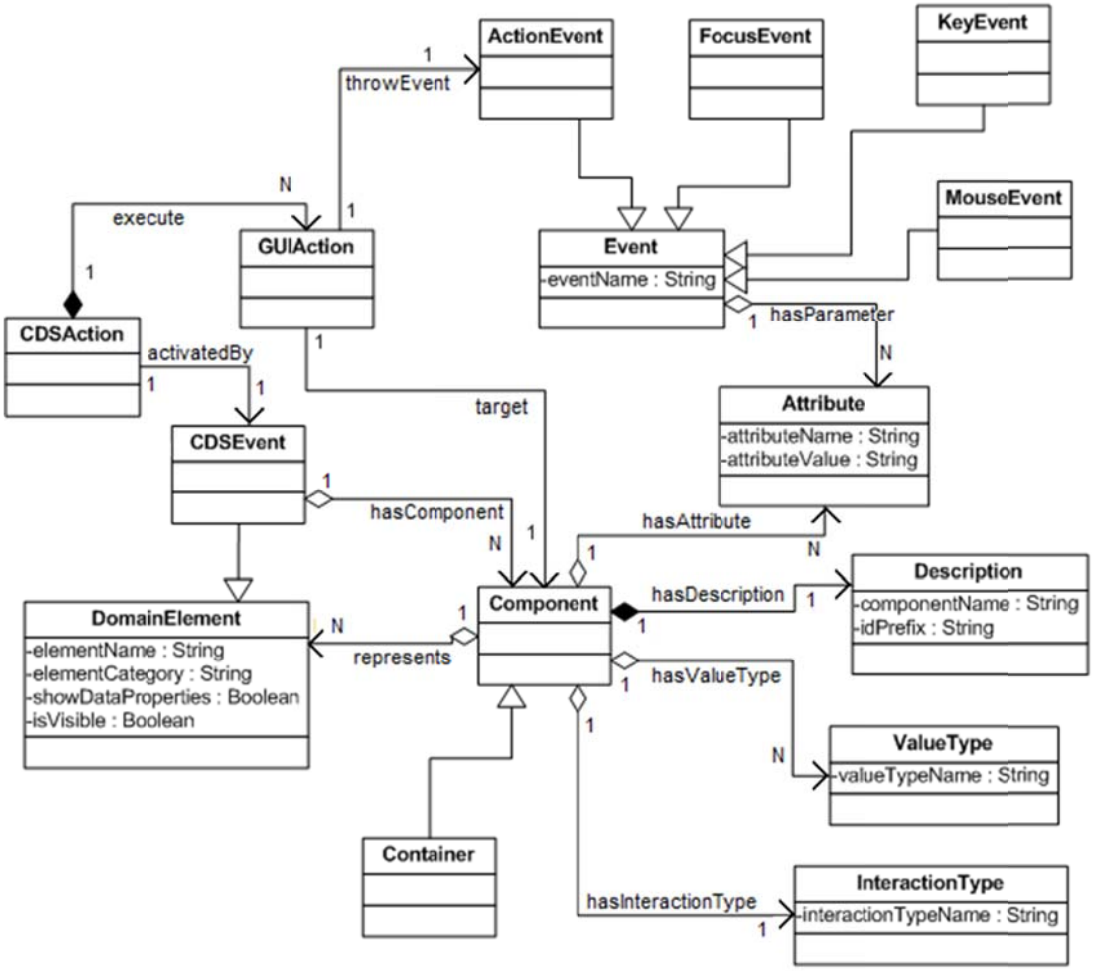

Figura 48 - Modelo ontológico de UI do MedViewGen (DUARTE, 2011)